

Стивен Дьюхерст

C++ СВЯЩЕННЫЕ ЗНАНИЯ



C++ Common Knowledge

Essential Intermediate Programming

Stephen C. Dewhurst

Addison-Wesley



C++
Священные знания

Основы успеха для будущих профессионалов

Издание исправленное

Стивен С. Дьюхерст



Санкт-Петербург — Москва
2012

Серия «Профессионально»

Стивен С. Дьюхерст

C++. Священные знания

Перевод Н. Шатохиной

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*Н. Лощинин*
Редактор	*В. Овчинников*
Художник	*В. Гренда*
Корректор	*С. Минин*
Верстка	*Д. Орлова*

Дьюхерст С.

C++. Священные знания. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 240 с., ил.

ISBN 978-5-93286-095-3

Стивен Дьюхерст, эксперт по C++ с более чем 20-летним опытом применения C++ в различных областях, рассматривает важнейшие, но зачастую неправильно понимаемые темы программирования и проектирования на C++, отсеивая при этом ненужные технические тонкости. В один тонкий том Стив уместил то, что он и его рецензенты, опытные консультанты и авторы, считают самым необходимым для эффективного программирования на C++.

Книга адресована тем, кто имеет опыт программирования на C++ и испытывает необходимость быстро повысить свое знание C++ до профессионального уровня. Издание полезно и квалифицированным программистам на C или Java, имеющим небольшой опыт проектирования и разработки сложного кода на C++ и склонным программировать на C++ в стиле Java.

ISBN 978–5–93286–095–3
ISBN 0–321–32192–8 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 26.12.2011. Формат 70x90 1/16 .
Печать офсетная. Объем 15 печ. л.

Оглавление

Отзывы о книге

«Мы живем во время, когда, как это ни удивительно, только начинают появляться самые лучшие печатные работы по C++. Эта книга – одна из них. Хотя C++ стоит в авангарде нововведений и продуктивности разработки программного обеспечения более двух десятилетий, только сейчас мы достигли полного его понимания и использования. Данная книга – это один из тех редких вкладов, ценный как для практиков, так и для теоретиков. Это не трактат тайных или академических знаний. Скорее, книга дает доскональное описание известных, казалось бы, вещей, недопонимание которых рано или поздно даст о себе знать. Очень немногие овладели C++ и проектированием программного обеспечения так, как это сделал Стив. Практически никто не обладает такой рассудительностью и спокойствием, когда речь идет о разработке программного обеспечения. Он знает, что необходимо знать, поверьте мне. Когда он говорит, я всегда внимательно слушаю. Советую и вам делать так же. Вы (и ваши заказчики) будете благодарны за это.»

Чак Эллисон (Chuck Allison),
редактор журнала «The C++ Source»

«Стив обучил меня C++. Это было в далеком 1982 или 1983. Я думаю, он тогда только вернулся с последипломной практики, которую проходил вместе с Бьерном Страуструпом (Bjarne Stroustrup) [создателем C++] в научно-исследовательском центре Bell Laboratories. Стив – один из невоспетых героев, стоящих у истоков. Все, что им написано, я отношу к обязательному и первоочередному чтению. Эта книга легко читается, она вобрала в себя большую часть обширных знаний и опыта Стива. Настоятельно рекомендую с ней ознакомиться.»

Стен Липман (Stan Lippman),
соавтор книги «C++ Primer, Fourth Edition»

«Я приветствую обдуманный профессиональный подход коротких и толковых книг.»

Мэтью П. Джонсон (Matthew P. Johnson),
Колумбийский университет

«Согласен с классификацией программистов [сделанной автором]. В своей практике разработчика мне доводилось встречаться с подобными людьми. Эта книга должна помочь им заполнить пробел в образовании... Я думаю, данная книга дополняет другие, такие как «Effective C++» Скотта Мейерса (Scott Meyers). Вся информация в ней представлена кратко и проста для восприятия.»

Моутаз Кэмел (Moataz Kamel), ведущий разработчик
программного обеспечения, компания Motorola, Канада

«Дьюхерст написал еще одну очень хорошую книгу. Она необходима людям, использующим C++ (и думающим, что они уже все знают о C++).»

Кловис Тондо (Clovis Tondo),
соавтор книги «C++ Primer Answer Book»

Предисловие

Успешность книги определяется не ее содержанием,
а тем, что осталось вне ее рассмотрения.

Марк Твен

...просто, насколько возможно, но не проще.

Альберт Эйнштейн

...писатель, ставящий под вопрос
умственные способности читателя,
вообще не писатель, а просто прожектер.

Е. В. Уайт

Заняв должность редактора ныне закрывшегося журнала «C++ Report», непоседливый Герб Саттер (Herb Sutter) предложил мне вести колонку и выбрать тему на мое усмотрение. Я согласился и решил назвать колонку «Общее знание». Герб представлял эту колонку как «систематический обзор основных профессиональных знаний, которыми должен располагать каждый практикующий программист на C++». Однако, сделав пару выпусков в этом ключе, я заинтересовался методиками метапрограммирования шаблонов, и темы, рассматриваемые в «Общем знании», с этого момента стали далеко не такими «общими».

Проблема индустрии программирования на C++, обусловившая выбор темы, осталась. В своей практике преподавателя и консультанта я сталкиваюсь со следующими типами личностей:

- программисты, в совершенстве владеющие C, но имеющие только базовые знания C++ и, возможно, некоторую неприязнь к нему;

- талантливые новобранцы, прямо с университетской скамьи, имеющие глубокую теоретическую подготовку в C++, но небольшой опыт работы с этим языком;
- высококвалифицированные Java-программисты, имеющие небольшой опыт работы с C++ и склонные программировать на C++ в стиле Java;
- программисты на C++ с опытом сопровождения готовых приложений на C++, у которых не было необходимости изучать что-либо сверх того, что требуется для сопровождения кода.

Хотелось бы сразу перейти к делу, но многим из тех, с кем я сотрудничаю или кого обучаю, требуется предварительное изучение различных возможностей языка программирования C++, шаблонов и методик написания кода. Хуже того, я подозреваю, что большая часть кода на C++ написана в неведении о некоторых из этих основ и, следовательно, не может быть признана специалистами качественным продуктом.

Данная книга направлена на решение этой всепроникающей проблемы. В ней основные знания, которыми должен обладать каждый профессиональный программист на C++, представлены в такой форме, что могут быть эффективно и четко усвоены. Значительная часть материала доступна в других источниках или уже входит в неписаные рекомендации, известные всем экспертам в C++. Преимущество книги в том, что вся информация, в отборе которой я руководствовался своим многолетним опытом преподавания и консультирования, расположена компактно.

Очень может быть, что самое важное в шестидесяти трех коротких темах, составляющих данную книгу, заключается в том, что осталось вне рассмотрения, а не в том, что они содержат. Многие из этих вопросов потенциально очень сложны. Если бы автор не был осведомлен об этих сложностях, то недостоверные сведения ввели бы читателя в заблуждение; в то же время профессиональное обсуждение вопроса с полным представлением всех сложных аспектов могло бы его запутать. Все слишком сложные вопросы были отсеяны. Надеюсь, то, что осталось, представляет собой квинтэссенцию знаний, необходимых для продуктивного программирования на C++. Знатоки заметят, что я не рассматриваю некоторые вопросы, интересные и даже важные с теоретической точки зрения, незнание которых, однако, как правило, не влияет на способность читать и писать код на C++.

Другим толчком для написания данной книги стала моя беседа с группой известных экспертов в C++ на одной из конференций. Все они были настроены мрачно, считая современный C++ слишком сложным и потому недоступным для понимания «среднестатистическим» программистом. (Конкретно речь шла о связывании имен в контексте шаблонов и пространств имен. Да, если ты раздражаешься по такому поводу, значит, пора

больше общаться с нормальными людьми.) Поразмыслив, должен сказать, что наша позиция была высокомерной, а пессимизм – неоправданным. У нас, «экспертов», нет таких проблем, и программировать на C++ так же просто, как говорить на (намного более сложном) естественном языке, даже если ты не можешь схематически представить глубинную структуру каждого своего высказывания. Основная идея данной книги: даже если полное описание нюансов конкретной возможности языка выглядит устрашающе, ее рутинное использование вполне может быть бесхитростным и естественным.

Возьмем перегрузку функций. Полное описание занимает немалую часть стандарта и одну или несколько глав многих учебников по C++. И тем не менее, столкнувшись с таким кодом

```
void f( int );
void f( const char * );
//…
f( "Hello" );
```

любой практикующий программист на C++ безошибочно определит, какая `f` вызывается. Знание всех правил разрешения вызова перегруженной функции полезно, но очень редко бывает необходимым. То же самое можно сказать о многих других якобы сложных областях и идиомах языка программирования C++.

Сказанное не означает, что вся книга проста; «она проста настолько, насколько это возможно, но не проще». В программировании на C++, как в любой другой достойной внимания интеллектуальной деятельности, многие важные детали нельзя уместить на карточке картотеки. Более того, эта книга не для «чайников». Я чувствую огромную ответственность перед теми, кто тратит свое драгоценное время на чтение моих книг. Я уважаю этих людей и стараюсь общаться с ними как с коллегами. Нельзя писать для профессионалов, как для восьмиклассников, потому что это чистая профанация.

Многие темы книги опровергают заблуждения, с которыми я многократно сталкивался и на которые просто надо обратить внимание (например, посвященные порядку областей видимости при поиске функции-члена, разнице между переопределением и перегрузкой). В других обсуждаются вопросы, знание которых постепенно становится обязательным для профессионалов C++, но нередко ошибочно считающиеся сложными и потому замалчиваемые (например, частичная специализация шаблонов класса и параметры-шаблоны шаблонов). Я был раскритикован экспертами, рецензировавшими рукопись этой книги, за то, что слишком много внимания (примерно треть книги) уделил вопросам шаблонов, которые не относятся к общим знаниям. Однако каждый из этих специалистов указал

один, два и более вопросов по шаблонам, которые, по их мнению, следовало включить в книгу. Примечательно то, что предлагаемые ими вопросы немного перекрывались. В итоге каждый из рассмотренных мною аспектов шаблонов получил, по крайней мере, одного сторонника.

В этом и состояла главная трудность отбора тем для данной книги. Вряд ли найдутся читатели, абсолютно не сведущие ни в одном из рассматриваемых вопросов. Скорее всего, некоторым из них будет хорошо знаком весь материал. Очевидно, что если читатель не знает конкретной темы, то ему было бы полезно (я так думаю) изучить ее. Но рассмотрение даже знакомого вопроса под новым углом способно развеять некоторые заблуждения или помочь глубже понять вопрос. А более опытным программистам на C++ книга поможет сберечь драгоценное время, которое они нередко вынуждены (как уже отмечалось) отрывать от своей работы, без конца отвечая на одни и те же вопросы. Я предлагаю интересующимся сначала прочитать книгу, а потом спрашивать, и думаю, что это поможет направить усилия специалистов на решение сложных проблем, туда, где это действительно необходимо.

Сначала я пытался четко сгруппировать шестьдесят три темы в главы, но испытал неожиданные затруднения. Темы стали объединяться самовольно – иногда вполне предсказуемо, а иногда совершенно неожиданным образом. Например, темы, посвященные исключениям и управлению ресурсами, образуют довольно естественную группу. Взаимосвязь тем «Запросы возможностей», «Смысл сравнения указателей», «Виртуальные конструкторы и Прототип», «Фабричный метод» и «Ковариантные возвращаемые типы» хотя и тесна, но несколько неожиданна. «Арифметику указателей» я решил представить вместе с «Умными указателями», а не с изложенным ранее материалом по указателям и массивам. Вместо того чтобы навязать жесткую группировку тем по главам, я предоставил читателю свободу ассоциации. Конечно, между темами, которые могли бы быть расположены в простом линейном порядке, существует масса других взаимосвязей, поэтому в темах делается множество внутренних ссылок друг на друга. Это разбитое на группы, но взаимосвязанное сообщество.

Хотя основной идеей книги является краткость, обсуждение темы иногда включает дополнительные детали, не касающиеся непосредственно рассматриваемого вопроса. Эти подробности не всегда относятся к теме дискуссии, но читателю сообщается об определенной возможности или методике. Например, шаблон `Heap`, появляющийся в нескольких темах, мимоходом информирует читателя о существовании полезных, но редко рассматриваемых алгоритмов STL работы с кучей. Обсуждение синтаксиса размещения `new` представляет техническую базу сложных методик управления буферами, широко используемых стандартной библиотекой. Также везде, где это казалось уместным, я пытался включить обсуждение

вспомогательных вопросов в рассмотрение конкретной проблемы. Поэтому тема «Методика RAII» содержит краткое обсуждение порядка вызова конструктора и деструктора, в теме «Логический вывод аргументов шаблона» обсуждается применение вспомогательных функций для специализации шаблонов классов, а «Присваивание и инициализация – не одно и то же» включает рассмотрение вычислительных конструкторов. В этой книге запросто могло бы быть в два раза больше тем. Но и группировка самих тем, и связь вспомогательных вопросов с конкретной темой помещают проблему в контекст и помогают читателю эффективно усваивать материал.

Я с неохотой включил несколько вопросов, которые не могут быть рассмотрены корректно в формате коротких тем, принятом в данной книге. В частности, вопросы шаблонов проектирования и проектирования стандартной библиотеки шаблонов представлены смехотворно кратко и неполно. Но все же они вошли сюда, просто чтобы положить конец некоторым распространенным заблуждениям, подчеркнуть свою важность и подтолкнуть читателя к более глубокому изучению.

Традиционные примеры являются частью нашей культуры программирования, как истории, которые рассказывают на семейных праздниках. Поэтому здесь появляются `Shape`, `String`, `Stack` и далее по списку. Всеобщее понимание этих базовых примеров обеспечивает тот же эффект при общении, что и паттерны проектирования. Например, «Предположим, я хочу вращать `Shape`, кроме...» или «При конкатенации двух `String`...». Простое упоминание обычного примера определяет направление беседы и устраняет необходимость длительного обсуждения предпосылок.

В отличие от моей предыдущей книги, здесь я пытаюсь избегать критики плохих практик программирования и неверного использования возможностей языка C++. Пусть этим занимаются другие книги, лучшие из которых я привел в списке литературы. (Однако мне не вполне удалось избежать менторского тона; некоторые плохие практики программирования просто необходимо упомянуть, хотя бы вскользь.) Цель данной книги – рассказать читателю о технической сущности производственного программирования на C++ максимально эффективным способом.

Стивен С. Дьюхерст,
Карвер, Массачусетс, январь 2005

Благодарности

Питер Гордон (Peter Gordon) – редактор от Бога и экстраординарная личность – удивительно долго выдерживал мое нытье по поводу состояния образования в сообществе разработчиков на C++, пока не предложил мне самому попытаться что-то сделать. В результате появилась эта книга. Ким Бодигхеймер (Kim Boedigheimer) как-то сумел проконтролировать весь проект, ни разу не оказав существенного давления на автора.

Опытные технические редакторы – Мэтью Джонсон (Matthew Johnson), Моутаз Кэмел (Moataz Kamel), Дэн Сакс (Dan Saks), Кловис Тондо (Clovis Tondo) и Мэтью Вилсон (Matthew Wilson) – нашли несколько ошибок и множество погрешностей в языке рукописи, чем помогли сделать эту книгу лучше. Но я упрямец и последовал не всем рекомендациям, поэтому любые ошибки или погрешности языка – полностью моя вина.

Некоторые материалы данной книги появлялись в немного иной форме в моей колонке «Общее знание» журнала «C/C++ Users Journal», и многое из представленного здесь можно найти в веб-колонке «Once, Weakly» по адресу *semantics.org*. Я получил множество развернутых комментариев как на печатные, так и на опубликованные в сети статьи от Чака Эллисона (Chuck Allison), Аттилы Фахира (Attila Feher), Келвина Хенни (Kevlin Henney), Торстена Оттосена (Thorsten Ottosen), Дэна Сакса, Тери Слеттебо (Terje Slettebo), Герба Саттера (Herb Sutter) и Леора Золмана (Leor Zolman). Некоторые глубокие дискуссии с Дэном Саксом улучшили мое понимание разницы между специализацией и созданием экземпляра шаблона и помогли прояснить различие между перегрузкой и представлением перегрузки при ADL и поиске инфиксного оператора.

Я в долгу перед Брэндоном Голдфеддером (Brandon Goldfedder) за аналогию алгоритмов и шаблонов, проводимую в теме, посвященной шаблонам проектирования, и перед Кловисом Тондо за мотивирование и помощь

в поиске квалифицированных рецензентов. Мне повезло в течение многих лет вести курсы на базе книг Скотта Мейерса [5, 6, 7], что позволило из первых рук узнать, какую информацию обычно упускают учащиеся, использующие эти отражающие промышленный стандарт книги по C++ для специалистов среднего уровня. Эти наблюдения помогли сформировать набор тем данной книги. Работа Андрея Александреску (Andrei Alexandrescu) вдохновила меня на эксперименты с метапрограммированием шаблонов. А работа Герба Саттера и Джека Ривза (Jack Reeves), посвященная исключениям, помогла лучше понять их использование.

Я также хотел бы поблагодарить моих соседей и хороших друзей Дика и Джуди Ворд (Dick, Judy Ward), которые периодически выгоняли меня из-за компьютера, чтобы поработать на сборе урожая клюквы. Тому, чья профессиональная деятельность связана преимущественно с упрощенными абстракциями реальности, полезно увидеть, что убедить клюквенный куст плодоносить настолько же сложно, как все то, что может делать программист на C++ с частичной специализацией шаблона.

Сара Дж. Хьювинс (Sarah G. Hewins) и Дэвид Р. Дьюхерст (David R. Dewhurst), как всегда, обеспечили этому проекту одновременно и неоценимую поддержку, и крайне необходимые препятствия.

Мне нравится считать себя спокойным человеком с устойчивыми привычками, больше предрасположенным к спокойному созерцанию, чем громкому предъявлению требований. Однако, подобно тем, кто претерпевает трансформацию личности, оказавшись за рулем автомобиля, закончив рукопись, я стал совершенно другим человеком. Замечательная команда специалистов по модификации поведения издательства Addison-Wesley помогла мне преодолеть эти личностные проблемы. Чанда Лери-Коту (Chanda Leary-Coutu) с Питером Гордоном и Кимом Бодигхеймером работали над переводом моих разглагольствований в рациональные бизнес-предложения и утверждением их у сильных мира сего. Молли Шарп (Molly Sharp) и Джулия Нагил (Julie Nahil) не только превратили неудобный документ Word в лежащие перед вами аккуратные страницы, но и умудрились устранить множество недостатков рукописи, позволив мне при этом сохранить архаичную структуру своих предложений, необычный стиль и характерную расстановку переносов. Несмотря на мои постоянно меняющиеся запросы, Ричард Эванс (Richard Evans) смог уложиться в график и создать предметный указатель. Чути Прасертсис (Chuti Prasertsith) разработала великолепную обложку с клюквенными мотивами. Всем огромное спасибо.

Принятые обозначения

Как упоминалось в предисловии, в этой книге много перекрестных ссылок. При ссылке на какую-либо тему приводится и ее номер, и полное название, чтобы читателю не приходилось постоянно сверяться с оглавлением с целью понять, чему посвящена данная тема. Например, ссылка «64 „Ешьте свои овощи“» говорит нам, что тема под названием «Ешьте свои овощи» имеет порядковый номер 64.

Примеры кода выделены моноширинным шрифтом, чтобы их можно было отличить от остального текста. Примеры неверного или нерекомендуемого кода выделены серым фоном. Код без ошибок представлен без фона.

Тема 1 | Абстракция данных

Для программиста *тип данных* – это набор операций, а *абстрактный тип* – это набор операций с реализацией. Идентифицируя объекты в предметной области, мы прежде всего должны интересоваться, что можно сделать с помощью этого объекта, а не как он реализован. Следовательно, если в описании задачи присутствуют служащие, контракты и платежные ведомости, то выбранный язык программирования должен содержать типы `Employee` (Служащий), `Contract` (Контракт) и `PayrollRecord` (Платежная ведомость). Тем самым обеспечивается эффективная двусторонняя трансляция между предметной областью и областью решения. Код, написанный таким образом, содержит меньше «трансляционного шума», он проще и в нем меньше ошибок.

В языках программирования общего назначения, к которым относится C++, нет специальных типов, таких как `Employee`. Зато есть кое-что получше, а именно средства для создания сложных абстрактных типов данных. По сути, назначение абстрактного типа данных состоит в расширении языка программирования в соответствии с нуждами конкретной предметной области.

В C++ нет универсальной процедуры проектирования абстрактных типов данных. В этой области программирования все еще есть место для творчества и вдохновения. Однако самые успешные подходы подразумевают выполнение похожих шагов.

1. Типу присваивается описательное имя. Если найти имя трудно, значит, нет достаточной информации о том, что именно предполагается реализовывать. Необходим дополнительный анализ. Абстрактный тип данных должен представлять одно четко определенное понятие, и имя этого понятия должно быть очевидным.

2. Перечисляются операции, которые может осуществлять тип. Абстрактный тип данных определяется тем, что можно сделать с его помощью. Нельзя забывать об инициализации (конструкторы), очистке ресурсов (деструктор), копировании (операции копирования) и преобразованиях (неявные конструкторы с одним аргументом и операторы преобразования). Ни в коем случае нельзя просто определять набор операций get/set над элементами данных реализации. Это не абстракция данных, а лень и недостаток воображения.
3. Проектируется интерфейс типа. Как говорит Скотт Мейерс (Scott Meyers), тип должен «помогать программировать правильно и мешать программировать неправильно». Абстрактный тип данных расширяет язык программирования. По сути, надо спроектировать язык. Попробуйте поставить себя на место пользователя этого типа и напишите немного кода, задействовав свой интерфейс. Правильная конструкция интерфейса – это вопрос как психологии и взаимопонимания, так и проявления технического мастерства.
4. Тип реализуется. Реализация не должна оказывать влияние на интерфейс типа. Должен реализовываться контракт, оговоренный интерфейсом типа. Необходимо помнить, что реализации большинства абстрактных типов данных будут меняться намного чаще, чем их интерфейсы.

Тема 2 | Полиморфизм

В одних книгах по программированию полиморфизму приписывается мистический статус, другие о полиморфизме молчат вовсе, а на самом деле это простая и полезная концепция, поддерживаемая языком C++. Согласно стандарту *полиморфный тип* – это класс, имеющий виртуальную функцию. С точки зрения проектирования *полиморфный объект* – это объект, имеющий более одного типа. И *полиморфный базовый класс* – это базовый класс, спроектированный для использования полиморфными объектами.

Рассмотрим тип финансового опциона `AmOption` (Американский опцион), представленный на рис. 2.1.

У объекта `AmOption` четыре типа: он одновременно выступает в ипостасях `AmOption`, `Option` (Опцион), `Deal` (Сделка) и `Priceable` (Подлежащий оплате).

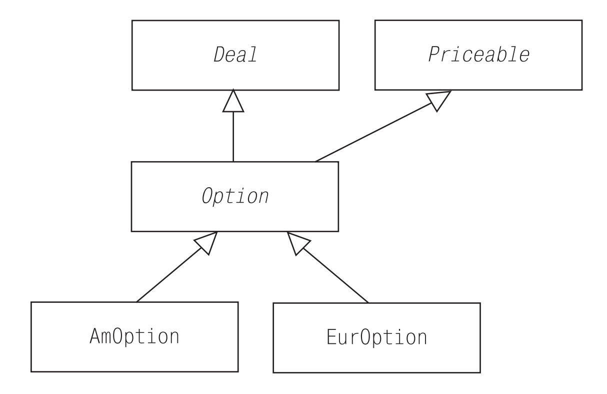


***Рис. 2.1.** Полиморфизм в иерархии финансового опциона. У Американского опциона четыре типа*

Поскольку тип – это набор операций (см. темы 1 «*Абстракция данных*» и 27 «*Запросы возможностей*»), с объектом `AmOption` можно работать посредством одного из его четырех интерфейсов. Это означает, что объектом `AmOption` может управлять код, написанный для интерфейсов `Deal`, `Priceable` и `Option`. Таким образом, реализация `AmOption` может применять и повторно использовать весь этот код. Для полиморфного типа, такого как `AmOption`, самым важным из наследуемого от базовых классов являются интерфейсы, а не реализации. Часто бывает желательно, чтобы базовый класс не включал ничего, кроме интерфейса (см. тему 27 «*Запросы возможностей*»).

Конечно, здесь есть своя тонкость. Чтобы эта иерархия классов работала, полиморфный класс должен уметь замещать любой из своих базовых классов. Другими словами, если универсальный код, написанный для интерфейса `Option`, получает объект `AmOption`, этот объект должен вести себя как `Option`!

Речь идет не о том, что `AmOption` должен вести себя идентично `Option`. (Прежде всего, многие операции базового класса `Option` нередко представляют собой чисто виртуальные функции без реализации.) Лучше рассматривать полиморфный базовый класс (`Option`) как контракт. Базовый класс дает определенные обещания пользователям его интерфейса: сюда входят твердые синтаксические гарантии вызова определенных функций-членов с определенными типами аргументов, а также обещания, которые сложнее проверить, касающиеся того, что на самом деле произойдет при вызове конкретной функции-члена. Конкретные производные классы, такие как `AmOption` и `EurOption` (Европейский опцион), представляют собой субконтракторы, которые реализуют контракт, устанавливаемый классом `Option` с его клиентами, как показано на рис. 2.2.

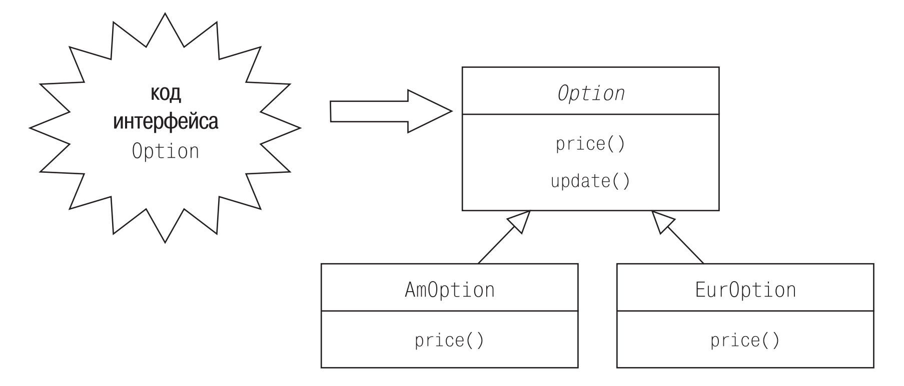


Рис. 2.2. *Полиморфный контрактор и его субконтракторы. Базовый класс Option определяет контракт*

Например, если в классе Option есть чисто виртуальная функция-член price (цена), вычисляющая текущее значение Option, то оба класса, AmOption и EurOption, должны реализовывать эту функцию. Очевидно, что в этих двух типах Option ее поведение не будет идентичным, но она должна вычислять и возвращать цену, а не звонить по телефону или распечатывать файл.

С другой стороны, если вызвать функцию price двух разных интерфейсов *одного* объекта, результат должен быть одним и тем же. По сути, любой вызов должен быть связан с одной и той же функцией:

```
AmOption *d = new AmOption;
Option *b = d;
d->price();   // если здесь вызывается AmOption::price...,
b->price();   // ...то же самое должно происходить здесь!
```

Это логично. (Просто удивительно, но углубленные аспекты объектно-ориентированного программирования по большей части есть не что иное, как проявление здравого смысла, скрытое завесой синтаксиса.) Вопросы: «Каково текущее значение этого Американского опциона?» и «Каково текущее значение этого опциона?», по моему мнению, требовали бы одного ответа.

Аналогичные рассуждения, конечно же, применимы и к невиртуальным функциям объектов:

```
b->update();   // если здесь вызывается Option::update...,
d->update();   // ... то же самое должно происходить здесь!
```

Контракт, предоставляемый базовым классом, – это то, что позволяет «полиморфному» коду интерфейса базового класса работать с конкретными опционами, оставаясь при этом в полезном неведении об их существовании. Иначе говоря, полиморфный код может управлять объектами AmOption и EurOption, рассматривая их как объекты класса Option. Различные конкретные типы могут добавляться или удаляться без всякого воздействия на универсальный код, который знает только о базовом классе Option. Если в какой-то момент появится AsianOption (Азиатский опцион), полиморфный код, знающий только Option, сможет работать с ним в блаженном неведении о его конкретном типе. И если позже этот класс исчезнет, его отсутствие не будет замечено.

Справедливо и то, что конкретным типам опционов, таким как AmOption и EurOption, необходимо знать только о базовых классах, чьи контракты они реализуют. Они не зависят от изменений универсального кода. В принципе базовый класс может знать только о себе. Практически конструкция его интерфейса будет учитывать требования предполагаемых пользователей. Он должен проектироваться таким образом, чтобы произ-

водные классы могли без труда проследить и реализовать его контракт (см. тему 22 «*Шаблонный метод*»). Однако у базового класса не должно быть никакой конкретной информации о любом из его производных классов, потому что это неизбежно усложнит добавление или удаление производных классов в иерархии.

В объектно-ориентированном проектировании, как и в жизни, действует правило «много будешь знать – скоро состаришься» (см. также темы 29 «*Виртуальные конструкторы и Прототип*» и 30 «*Фабричный метод*»).

Тема 3 | Паттерны проектирования

У тех, кто еще не знаком с паттернами проектирования, после краткого обзора данной темы может создаться впечатление, что паттерны проектирования – это либо маркетинговый прием, либо какая-то простая методика написания кода, либо забава ученых, которым на самом деле надо почаще выходить прогуляться. Хотя во всем этом и есть некоторая доля истины, паттерны проектирования – это важнейший компонент инструментария профессионального программиста на C++.

Паттерн проектирования – это многократно применяемая архитектурная конструкция, предоставляющая решение общей проблемы проектирования в рамках конкретного контекста и описывающая результат этого решения. Паттерн проектирования – это больше, чем простое описание техники. Это именованный блок конструкторской мудрости, накопленной в результате успешного опыта, написанный таким образом, что он может без труда тиражироваться и повторно использоваться. Паттерны – это средство обмена информацией между разработчиками.

Паттерны проектирования обладают двумя важными практическими свойствами. Во-первых, они описывают проверенную успешную методику проектирования, которая может быть настроена соответственно контексту новых ситуаций проектирования. Во-вторых, что еще важнее, выбор определенного паттерна говорит не только о выбранной технике, но и о причинах и следствиях ее применения.

Здесь нет ничего нового. Рассмотрим аналогию из области алгоритмов. (Алгоритмы не являются ни паттернами проектирования, ни «паттернами кода». Они всего лишь алгоритмы, поэтому мы и говорим об аналогии.) Возьмем следующее высказывание, с которым я мог бы обратиться к коллегам: «У меня есть неотсортированная последовательность, по которой необходимо проводить многократный поиск. Поэтому я собираюсь

осуществить быструю сортировку и каждую операцию поиска выполнять с помощью двоичного поиска». Возможность употребить термины «быстрая сортировка» и «двоичный поиск» имеет неоценимое значение не только при проектировании, но и при общении с коллегами. Когда я говорю «быстрая сортировка», мой коллега знает, что сортируемая последовательность находится в структуре с произвольным доступом, которая, вероятнее всего, будет сортироваться до $O(n \lg_2 n)$ раз, и что элементы последовательности могут сравниваться с помощью оператора, подобного оператору «меньше чем». Когда я говорю «двоичный поиск», мой коллега знает (даже если я ранее не упоминал о «быстрой сортировке»), что последовательность уже отсортирована, что я буду выявлять интересующий элемент с помощью $O(\lg_2 n)$ сравнений и что для сравнения элементов последовательности есть соответствующая операция. Общеизвестность и стандартный словарь стандартных алгоритмов обеспечивают возможность не только эффективного документирования, но и эффективной критики. Например, если бы я запланировал проводить эту процедуру поиска и сортировки для однонаправленного списка, мой коллега немедленно ухмыльнулся бы и сказал, что я не могу применять быструю сортировку, да и двоичный поиск мне вряд ли нужен.

«Допаттерная» эпоха в объектно-ориентированном проектировании характеризовалась тем, что не было тогда ни радостей эффективного документирования и общения, ни этих действенных саркастических улыбок. Приходилось пускаться в подробные описания своих проектов, мирясь с неэффективностью и неточностью, свойственным таким методам работы. И дело не в том, что не было правильных методик объектно-ориентированного проектирования, – не было единой терминологии, способной сделать эти методики доступными всему сообществу разработчиков. Паттерны проектирования решают эту проблему. Теперь объектно-ориентированные проекты могут описываться так же эффективно и однозначно, как и алгоритмические.

Например, когда мы видим, что в проекте применяется паттерн Мост (Bridge), мы знаем, что на простом механическом уровне реализация абстрактного типа данных была разделена на интерфейсный класс и класс реализации. Кроме того, известно, что это было сделано с целью строгого разделения интерфейса и реализации, чтобы изменения в реализации не отражались на пользователях интерфейса. Мы также знаем о существовании издержек времени выполнения, связанных с этим разделением, о том, как должен быть организован исходный код абстрактного типа данных, и о многих других деталях. Имя паттерна – это эффективный, однозначный дескриптор огромного объема информации и опыта. Правильное и аккуратное применение паттернов и их терминологии при проектировании и документировании делает код и проекты более понятными.

Поклонники паттернов иногда описывают паттерны проектирования как своего рода литературный жанр (это действительно так), имеющий определенную формальную структуру. В ходу несколько общих вариантов, но в каждом есть четыре основные части.

Во-первых, паттерн проектирования должен иметь однозначное имя. Например, термин «обертка» («wrapper») в данном случае не годится, потому что он уже широко распространен и имеет десятки значений. Употребление подобного термина в качестве имени паттерна проектирования приведет только к путанице и неправильному толкованию.

Поэтому различные техники проектирования, ранее известные под именем «обертка», сейчас обозначаются именами паттернов Мост (Bridge), Стратегия (Strategy), Фасад (Facade), Объектный адаптер (Object Adapter) и рядом других. Точное имя паттерна имеет явное преимущество перед расплывчатым термином, так же как термин «двоичный поиск» намного определеннее и полезнее, чем «поиск».

Во-вторых, описание паттерна должно определять задачу, на решение которой направлен паттерн. Это описание может быть относительно обширным или кратким.

В-третьих, описание паттерна определяет решение задачи. В зависимости от ее постановки решение может быть изложено либо на довольно высоком уровне, либо на относительно низком. Но в любом случае оно должно быть достаточно общим, чтобы оставалась возможность его настройки соответственно различным контекстам, в которых может возникнуть данная проблема.

В-четвертых, описание паттерна определяет и последствия применения паттерна к контексту. Как изменится контекст после применения паттерна – в лучшую или в худшую сторону?

Сделает ли знание паттернов плохого проектировщика хорошим? Здесь будет уместна другая аналогия. Вспомним один из тех зубодробительных курсов по математике, которые вам, может быть, приходилось изучать и сдавать в конце мучительный экзамен, доказывая теоремы. Как пережить этот кошмар? Очевидный способ состоит в том, чтобы быть гением. Начиная с самых основ, вы надстраиваете «здание» всего раздела математики и в конце концов доказываете теоремы. Более надежный подход предусматривает необходимость запомнить и усвоить много теорем из данной области математики, напрячь все отпущенные природой способности, призвать на помощь вдохновение и/или удачу, отыскать соответствующие вспомогательные теоремы и строить на их основе логические цепочки, чтобы доказывать новые теоремы.

Такой подход предпочтительнее даже для нашего воображаемого гения, поскольку на базе готовых теорем легче строить доказательство и объяснять его простым смертным. Знание вспомогательных теорем, конечно, не гарантирует слабому математику сдачу экзамена, но, по крайней мере, позволит ему понять доказательство.

Так и разработка сложных объектно-ориентированных проектов с нуля весьма утомительна. Еще сложнее донести до аудитории смысл окончательной конструкции. Роль паттернов проектирования в создании объектно-ориентированной конструкции сродни роли, которую в математике играют вспомогательные теоремы при доказательстве новой. Паттерны проектирования часто описываются как *микроархитектуры*, которые можно сочетать с другими паттернами для создания новой архитектуры. Конечно, выбор соответствующих паттернов и эффективное их сочетание требуют опыта проектирования и природных способностей. Однако даже ваш руководитель сможет понять окончательную конструкцию, если он знаком с паттернами.

Тема 4 | Стандартная библиотека шаблонов

Краткое описание стандартной библиотеки шаблонов (Standard Template Library – STL) не может представить всех ее возможностей в проектировании. Все нижеизложенное представляет собой лишь затравку, призванную вдохновить читателя на глубокое изучение STL.

STL не совсем библиотека. Это дарованная свыше идея и набор соглашений.

STL состоит из компонентов трех основных типов: контейнеров, алгоритмов и итераторов. Контейнеры содержат и организуют элементы. Алгоритмы осуществляют операции. Итераторы служат для доступа к элементам контейнера. Ничего нового, во многих традиционных библиотеках эти компоненты есть, и многие традиционные библиотеки реализованы с помощью шаблонов. «Божественная» идея STL состоит в том, что контейнеры и алгоритмы, работающие с ними, могут и не знать друг о друге. Это преимущество достигается с помощью итераторов.

Итератор имеет сходство с указателем. (По сути, указатели – это одна из разновидностей итераторов STL.) Как и указатель, итератор может ссылаться на элемент последовательности, может быть разыменован для получения значения объекта, на который он ссылается, и может предоставлять интерфейс, аналогичный простому указателю для обращения к разным элементам последовательности. Итераторы STL могут быть реализованы через предопределенные указатели или определяемые пользователями типы классов, перегружающие соответствующие операторы, чтобы получить синтаксис, аналогичный предопределенному указателю (см. тему 42 «*Умные указатели*»).

Контейнер STL – это абстракция структуры данных, реализованная как шаблон класса. Как и структуры данных, разные контейнеры организу-

ют свои элементы по-разному для оптимизации доступа или работы с ними. STL определяет семь (или, учитывая `string`, восемь) стандартных контейнеров и предлагает еще несколько широко распространенных нестандартных контейнеров.

Алгоритм STL – это абстракция функции, реализованная как шаблон функции (см. тему 60 «*Универсальные алгоритмы*»). Большинство алгоритмов STL работают с одной или более последовательностями значений, где последовательность определяется упорядоченной парой итераторов. Первый итератор указывает на первый элемент последовательности, второй – на следующий после последнего элемент последовательности (не на последний элемент). Если итераторы указывают на одну и ту же ячейку памяти, они определяют пустую последовательность.

Итераторы – это механизм, с помощью которого организуется совместная работа контейнеров и алгоритмов. Контейнер может создавать пару итераторов, обозначающих последовательность его элементов (как всех, так и поддиапазона), а алгоритм производит операции с этой последовательностью. Таким образом, контейнеры и алгоритмы могут тесно сотрудничать, оставаясь в неведении друг о друге. (Положительный эффект неведения в профессиональном программировании на C++ – повторное использование. См. темы 2 «*Полиморфизм*», 30 «*Фабричный метод*», 19 «*Команды и Голливуд*» и 60 «*Универсальные алгоритмы*».)

Кроме контейнеров, алгоритмов и итераторов, STL определяет ряд вспомогательных возможностей. Алгоритмы и контейнеры могут быть настроены с помощью указателей на функции и объектов-функций (см. тему 20 «*Объекты-функции STL*»), а эти объекты-функции могут адаптироваться и комбинироваться с помощью различных адаптеров объектов-функций.

Контейнеры также могут быть настроены с помощью адаптеров контейнеров, которые меняют интерфейс контейнера, превращая его в стек, очередь или очередь с приоритетами.

В STL широко применяются соглашения. Контейнеры и объекты-функции должны описываться через стандартный набор имен вложенных типов (см. темы 53 «*Встроенная информация о типе*», 54 «*Свойства*» и 20 «*Объекты-функции STL*»). И адаптеры контейнеров, и адаптеры объектов-функций требуют от функций-членов наличия определенных имен и определенной информации о типе. Алгоритмам необходимо, чтобы передаваемые в них итераторы могли поддерживать определенные операции и идентифицировать назначение этих операций. Если при использовании и расширении STL соглашение не соблюдается, то рушатся и все надежды. Неукоснительное соблюдение соглашений при работе с STL, наоборот, облегчает жизнь разработчика.

Соглашения STL не определяют деталей реализации, но накладывают ограничения эффективности на реализацию. Кроме того, поскольку STL – это библиотека шаблонов, большая часть оптимизации и настройки может происходить во время компиляции. Многие упомянутые выше соглашения, касающиеся присваивания имен и включения информации, приняты именно для обеспечения существенной оптимизации времени компиляции. Средний специалист, применяющий STL, обычно работает так же эффективно, как высококвалифицированный специалист, пишущий код вручную, и без труда опережает среднего специалиста или любую команду программистов, обходящихся без STL. Кроме того, в результате, как правило, получается более ясный и легкий в сопровождении код.

STL надо знать и применять как можно активнее.

Тема 5 | Ссылки – это псевдонимы, а не указатели

Ссылка – это второе имя объекта. Объект, который инициализировал ссылку, может фигурировать в коде как под своим именем, так под именем ссылки на него.

```
int a = 12;
int &ra = a;    // ra - другое имя a
--ra;           // a == 11
a = 10;         // ra == 10
int *ip = &ra;  // ip указывает на a
```

Ссылки часто путают с указателями, вероятно, потому, что компиляторы C++ часто реализуют ссылки как указатели. Однако ссылки – это не указатели, и ведут они себя по-разному.

Между ссылками и указателями существуют три основных различия: нулевых ссылок нет, все ссылки требуют инициализации и ссылка всегда установлена на объект, которым она инициализирована. В предыдущем примере ссылка `ra` будет ссылаться на `a` в течение всей его жизни. Все самые грубые ошибки применения ссылок проистекают из непонимания этих различий.

Некоторые компиляторы способны отлавливать явные попытки создания нулевой ссылки:

```
Employee &anEmployee = *static_cast<Employee *>(0); // ошибка!
```

Однако компилятор может не выявить менее очевидные попытки создания нулевой ссылки, что обусловит неопределенное поведение во время выполнения:

```
Employee *getAnEmployee();
//…
Employee &anEmployee = *getAnEmployee(); //вероятно, плохой код
if( &anEmployee ==0 )                    //неопределенное поведение
```

Если getAnEmployee возвращает нулевой указатель, поведение этого кода неопределенное. В данном случае лучше организовать хранение результата getAnEmployee при помощи указателя.

```
Employee *employee = getAnEmployee();
if( employee ) //...
```

Требование инициализации ссылки предполагает, что объект, на который она установлена, должен существовать на момент инициализации. Это важно, поэтому повторю: ссылка представляет собой псевдоним объекта, существовавшего до инициализации ссылки. Если ссылка инициализирована и установлена на конкретный объект, использовать ее для ссылки на другой объект нельзя. Ссылка связана с объектом, инициализировавшим ее, в течение всей своей жизни. По сути, после инициализации ссылка исчезает и далее просто существует еще одно имя инициализировавшего ее объекта. Это свойство замещения и является причиной, по которой ссылки зачастую удобны в качестве формальных аргументов функции. В следующей шаблонной функции swap (перестановка) формальные аргументы a и b становятся псевдонимами для фактических аргументов вызова:

```
template <typename T>
void swap( T &a, T &b ) {
   T temp(a);
   a = b;
   b = temp;
}
//...
int x = 1, y = 2;
swap( x, y ); // x == 2, y == 1
```

В приведенном выше вызове swap a – это псевдоним x, а b – псевдоним y на время выполнения вызова. Обратите внимание, что у объекта, на который установлена ссылка, может не быть имени. Таким образом, ссылка может применяться для присвоения подходящего имени безымянному объекту.

```
int grades[MAX];
//...
swap( grades[i], grades[j] );
```

После инициализации формальных аргументов a и b шаблона swap для реальных аргументов grades[i] и grades[j] соответственно с этими двумя бе-

зымянными элементами массива можно работать посредством псевдонимов a и b. Это свойство может использоваться для упрощения и оптимизации.

Рассмотрим следующую функцию, задающую конкретный элемент двумерного массива:

```
inline void set_2d( float *a, int m, int i, int j ) {
   a[i * m + j] = a[i * m + j] * a[i * m + i] + a[i * m + j];        // ой!
}
```

Строку с комментарием «ой!» можно заменить более простым вариантом со ссылкой, у которого есть дополнительное преимущество – он правильный. (Вы заметили ошибку? Я – только со второго раза.)

```
inline void set_2d( float *a, int m, int i, int j ) {
    float &r = a[i * m + j];
    r = r * r + r;
}
```

Ссылку на неконстанту нельзя инициализировать литералом или временным значением.

```
double &d = 12.3;                                    //ошибка!
swap( std::string("Hello"), std::string(", World")); // ошибки!
```

А на константу – можно:

```
const double &cd = 12.3;               //OK
template <typename T>
T add( const T &a, const T &b ) {
    return a + b;
}
//...
const std::string &greeting
    = add(std::string("Hello"),std::string(", World"));   //OK
```

Если ссылка на константу инициализирована литералом, то она указывает на временное местоположение, инициализированное литералом. Следовательно, cd фактически ссылается не на литерал 12.3, а на временный объект типа double, который был инициализирован значением 12.3. Ссылка greeting установлена на безымянное возвращаемое значение add типа string. Обычно временные объекты уничтожаются (т. е. покидают область видимости и для них вызывается деструктор) в конце выражения, в котором создаются. Однако когда временный объект используется для инициализации ссылки на константу, он существует столько, сколько существует ссылка, ссылающаяся на него.

Тема 6 | Массив как тип формального параметра

С формальными параметрами типа массив связаны некоторые проблемы. Главный сюрприз для новичка в C/C++ – отсутствие типа «массив» как формального параметра функции, потому что массив передается как указатель на его первый элемент.

```
void average( int ary[12] );            // формальный параметр типа int *
//...
int anArray[] = { 1, 2, 3 };            // массив из 3-х элементов
const int anArraySize =
    sizeof(anArray)/sizeof(anArray[0]);  // == 3
average( anArray );                     // допустимо!
```

Для автоматического перехода от массива к указателю придуман очаровательный термин – *разложение* (*decay*). Массив разлагается до указателя на его первый элемент. Кстати, то же самое происходит и с функциями. Аргумент функции разлагается до указателя на функцию, но, в отличие от массива, который теряет свою границу, у разлагающейся функции «хватает ума», чтобы сохранять типы своих аргументов и возвращаемый тип. (Обратите также внимание на правильное вычисление `anArraySize` во время компиляции, на которое не влияет изменение набора инициализаторов массива или типа элементов массива.)

Поскольку граница массива игнорируется в формальном параметре функции, то ее, как правило, лучше опустить. Однако если функция ожидает в качестве аргумента указатель на последовательность элементов (массив), а не указатель на один объект, наверное, лучше поступить так:

```
void average( int ary[] );   // формальный параметр по-прежнему int *
```

Если, наоборот, важно точное значение границы массива и требуется, чтобы функция принимала только массивы с конкретным числом элементов, можно рассмотреть формальный параметр ссылочного типа:

```
void average( int (&ary)[12] );
```

Теперь наша функция будет принимать только массивы целых размером 12 элементов.

```
average( anArray );            // ошибка! anArray размером int [3]!
```

Шаблоны могут кое-что обобщить:

```
template <int n>
void average( int (&ary)[n] ); // позволяет компилятору логически вывести n
```

но более традиционное решение – явная передача размера массива.

```
void average_n( int ary[], int size );
```

Конечно, можно объединить эти два подхода:

```
template <int n>
inline void average( int (&ary)[n] )
    { average_n( ary, n ); }
```

Из данного обсуждения должно быть понятно, что одна из основных трудностей в работе с массивами как с параметрами функций состоит в том, что размер массива необходимо задать явно в типе формального параметра, передать как отдельный параметр или обозначить *символом-ограничителем* в самом массиве (например, `'\0'` обозначает конец массива символов, образующих строку). Другая сложность в том, что независимо от того, как массив объявлен, с ним часто работают посредством указателя на его первый элемент. Если этот указатель передается в функцию как фактический параметр, то наш предыдущий трюк с объявлением ссылки в типе формального параметра не поможет.

```
int *anArray2 = new int[anArraySize];
//...
average( anArray2 ); // ошибка! нельзя инициализировать int(&)[n] типом int *
average_n( anArray, anArraySize ); // OK...
```

Поэтому часто в наиболее традиционных случаях работы с массивами лучше применить один из стандартных контейнеров (обычно `vector` или `string`), и, как правило, этот вариант должен рассматриваться в первую очередь. (См. также шаблон класса `Array` в теме 61 «*Мы создаем экземпляр того, что используем*».)

Объявление многомерных массивов в качестве формальных параметров в сущности не намного сложнее, чем объявление простых массивов, но на вид страшнее:

```
void process( int ary[10][20] );
```

Как и в одномерном случае, формальный параметр – это не массив, а указатель на первый элемент массива. Однако многомерный массив – это массив массивов, поэтому формальный параметр представляет собой указатель на массив (см. темы 17 «*Разбираемся с операторами объявления функций и массивов*» и 44 «*Арифметика указателей*»).

```
void process( int (*ary)[20] ); // указатель на массив, состоящий
                                // из 20 элементов типа int
```

Обратите внимание, что вторые (и последующие) границы не разлагаются, потому что в противном случае было бы невозможно применять арифметику указателей к формальным параметрам (см. тему 44 «*Арифметика указателей*»). Как уже отмечалось, лучше сообщить человеку, читающему программу, о том, что фактический параметр должен быть массивом:

```
void process( int ary[][20] );  // по-прежнему указатель, но теперь
                                // это более очевидно
```

Обработка аргументов типа многомерный массив нередко превращается в упражнение по низкоуровневому кодированию, когда программист берет на себя роль компилятора и занимается вычислением индексов:

```
void process_2d( int *a, int n, int m ) {    // a - это массив n на m
   for( int i = 0; i < n; ++i )
      for( int j = 0; j < m; ++j )
         a[i * m + j] = 0;                     // индекс вычисляется «вручную»!
}
```

И как обычно, иногда можно навести порядок посредством шаблона.

```
template <int n, int m>
inline void process( int (&ary)[n][m] )
   { process_2d( &ary[0][0], n, m ); }
```

Попросту говоря, формальные параметры типа массив – это сплошная головная боль. Поэтому работать с ними надо аккуратно.

Тема 7 | Константные указатели и указатели на константу

В повседневных разговорах программисты на C++ часто называют указатели на константу константными указателями. А зря, потому что это два совершенно разных понятия.

```
T *pt = new T;       // указатель на T
const T *pct = pt;   // указатель на константу T
T *const cpt = pt;   // константный указатель на T
```

Прежде чем вводить константные квалификаторы в описание указателя, необходимо решить, что вы хотите сделать константой: указатель, объект, на который он указывает, или и то и другое. В объявлении `pct` указатель не константа, но объект, на который он указывает, считается константой; т. е. константный квалификатор применяется к базовому типу `T`, а не к модификатору указателя `*`. В случае с `cpt` объявляется константный указатель на объект, не являющийся константой. Константный квалификатор применяется к модификатору указателя `*`, а не к базовому типу `T`.

Чтобы еще больше запутать синтаксис указателей и констант, определен произвольный порядок объявления спецификаторов (т. е. всего того, что в описании указателя предшествует первому модификатору `*`). Так, следующие два описания объявляют переменные одного и того же типа:

```
const T *p1;  // указатель на константу T
T const *p2;  // также указатель на константу T
```

Хотя первая форма более традиционна, многие эксперты C++ сейчас рекомендуют вторую форму на том основании, что она менее неоднозначна. Такое описание можно прочитать в обратном направлении, например

«указатель на константу T». В сущности, форму записи можно выбрать любую, главное быть последовательным и внимательным, избегая распространенной ошибки – путаницы между объявлением константного указателя и указателя на константу.

```
T const *p3;      // указатель на константу
T *const p4 = pt;  // константный указатель на неконстанту
```

Конечно, можно объявить константный указатель на константу.

```
const T *const cpct1 = pt;     // все константы
T const *const cpct2 = cpct1;  // то же самое
```

Обратите внимание, что часто ссылка предпочтительнее константного указателя:

```
const T &rct = *pt;  // а не const T *const
T &rt = *pt;         // а не T *const
```

Заметьте, что в некоторых предыдущих примерах можно было преобразовать указатель на неконстанту в указатель на константу. Например, можно было инициализировать pct (типа const T *) значением pt (типа T *). Дело в том, что при этом, попросту говоря, ничего плохого произойти не может. Посмотрим, что происходит, когда адрес неконстантного объекта копируется в указатель на константу (рис. 7.1).

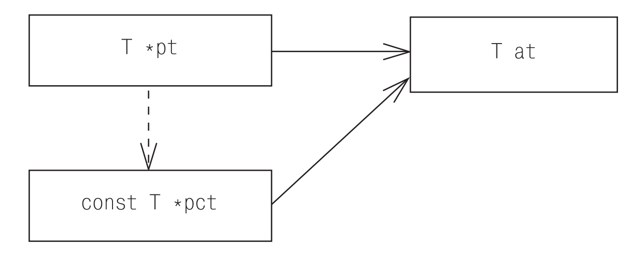


Рис. 7.1. *Указатель на константу может ссылаться на неконстантный объект*

Указатель на константу pct указывает на неконстантный T, что не вызывает никаких проблем. В действительности указатели (или ссылки) на константу очень часто ссылаются на неконстантные объекты:

```
void aFunc( const T *arg1, const T &arg2 );
//...
T *a = new T;
T b;
aFunc( a, b );
```

При вызове aFunc происходит инициализация arg1 объектом a и arg2 объектом b. Мы не утверждаем в связи с этим, что a указывает на константный объект или что b представляет собой константу. Мы утверждаем, что они будут трактоваться в рамках aFunc так, как будто являются константами, независимо от того, что они собой представляют на самом деле. Очень полезная возможность.

Обратное преобразование – из указателя на константу в указатель на неконстанту – недопустимо, потому что может быть опасным (рис. 7.2).

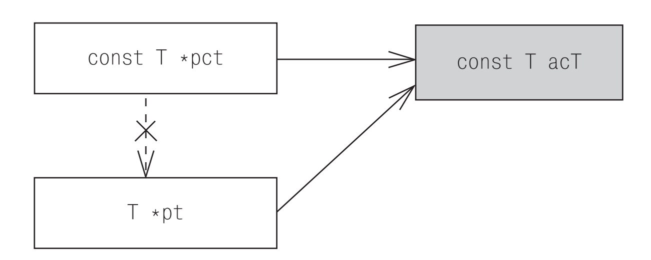


Рис. 7.2. *Указатель на неконстанту не может ссылаться на константный объект*

В данном случае pct может фактически указывать на объект, определенный как константный. Если бы можно было преобразовывать указатель на константу в указатель на неконстанту, pt мог бы использоваться для изменения значения acT.

```
const T acT;
pct = &acT;
pt = pct;   // ошибка, к счастью
*pt = acT;   // попытка изменить константный объект!
```

Стандарт C++ гласит, что такие присваивания приводят к неопределенным результатам. Что произойдет, точно не известно, но в любом случае ничего хорошего ожидать не приходится. Конечно, можно воспользоваться приведением и провести преобразование явно.

```
pt = const_cast<T *>(pct); // ошибки нет, но не рекомендуется

*pt = aT;          // попытка изменить константный объект!
```

Однако поведение операции присваивания остается неопределенным, если pt указывает на объект, который, как и acT, объявлен константным (см. тему 9 «*Новые операторы приведения*»).

Тема 8 | Указатели на указатели

Можно объявить указатель на указатель. В стандарте C++ такой указатель называется *многоуровневым*.

```
int *pi;       // указатель
int **ppi;     // двухуровневый указатель
int ***pppi;   // трехуровневый указатель
```

Хотя указатели с уровнем больше двух встречаются редко, есть два распространенных случая применения указателей на указатели. Первый – когда объявляется массив указателей.

```
Shape *picture[MAX];   // массив указателей на Shape
```

Поскольку имя массива разлагается в указатель на его первый элемент (см. тему 6 «*Массив как тип формального параметра*»), имя массива указателей также представляет собой указатель на указатель.

```
Shape **pic1 = picture;
```

Чаще всего такое применение можно увидеть в реализации класса, управляющего буфером указателей:

```
template <typename T>
class PtrVector {
   public:
      explicit PtrVector( size_t capacity )
         : buf_(new T *[capacity]), cap_(capacity), size_(0) {}
      //...
   private:
      T **buf_;         // указатель на массив указателей на T
      size_t cap_;      // емкость
      size_t size_;     // размер
```

```
};
//...
PtrVector<Shape> pic2( MAX );
```

Как подсказывает реализация PtrVector, указатели на указатели могут быть сложными. Лучше, чтобы они были скрытыми.

Изменение функцией значения переданного в нее указателя – второй общий случай использования многоуровневых указателей. Рассмотрим следующую функцию, которая меняет указатель так, чтобы он ссылался на следующий случай употребления символа в строке:

```
void scanTo( const char **p, char c ) {
   while( **p && **p != c )
       ++*p;
}
```

Первый аргумент scanTo – указатель, значение которого мы хотим изменить. Это означает необходимость передачи адреса указателя:

```
char s[] = "Hello, World!";
const char *cp = s;
scanTo( &cp, ',' ); // перемещает cp на первую запятую
```

Такое применение рационально в C. Однако в C++ проще, безопаснее и привычнее использовать как аргумент функции не указатель на указатель, а ссылку на указатель.

```
void scanTo( const char *&p, char c ) {
   while( *p && *p != c )
      ++p;
}
//...
char s[] = "Hello, World!";
const char *cp = s;
scanTo( cp, ',' );
```

В C++ практически всегда в функции ссылку на указатель следует предпочесть указателю на указатель.

Широко распространено следующее заблуждение: преобразования, применяемые к указателям, также применимы и к указателям на указатели. Это не так. К примеру, известно, что указатель на производный класс может быть преобразован в указатель на его открытый базовый класс:

```
Circle *c = new Circle;
Shape *s = c;       // замечательно...
```

Поскольку Circle (Круг) является объектом Shape (Фигура), указатель на Circle также является указателем на Shape. Однако указатель на указатель на Circle не является указателем на указатель на Shape:

```
Circle **cc = &c;
Shape **ss = cc;      // ошибка!
```

Такая же путаница часто возникает при работе с константами. Известно, что допускается преобразование указателя на неконстанту в указатель на константу (см. тему 7 «*Константные указатели и указатели на константу*»), но нельзя преобразовать указатель на указатель на неконстанту в указатель на указатель на константу:

```
char *s1 = 0;
const char *s2 = s1;  // OK...
char *a[MAX];         // также можно записать char **
const char **ps = a;  // ошибка!
```

Тема 9 | Новые операторы приведения

Приведение в старом стиле штука отчасти скрытная и коварная. Синтаксис этих операторов приведения таков, что нередко они остаются незамеченными в программном коде и могут приводить к большим неприятностям. Давайте определимся, что имеется в виду под «приведением в старом стиле». Очевидно, что оригинальный синтаксис C, согласно которому заключенный в скобки тип применяется к выражению, и есть приведение в старом стиле:

```
char *hopeItWorks = (char *)0x00ff0000;  // приведение в старом стиле
```

В C++ принят другой способ записи. В нем то же самое можно представить с помощью синтаксиса функционального стиля приведения:

```
typedef char *PChar;
hopeItWorks =
    PChar( 0x00ff0000 );     // функциональный/старый стиль приведения
```

Функциональный стиль приведения, может быть, выглядит приличнее, чем его ужасный предшественник, но он настолько же отвратителен. От них обоих надо бежать, как от чумы.

Настоящие программисты предпочитают новые операторы приведения, которые позволяют более строго оформлять в виде кода свои мысли и точнее формулировать сами мысли. Этих операторов всего четыре, и у каждого из них свое специальное назначение.

Оператор `const_cast` позволяет добавлять или удалять квалификаторы типов `const` и `volatile` из типа выражения:

```
const Person *getEmployee() { ... }
//...
```

```
Person *anEmployee = const_cast<Person *>(getEmployee());
```

В этом коде const_cast позволяет аннулировать квалификатор типа const в возвращаемом типе getEmployee. Такой же результат можно было бы получить, применив приведение в старом стиле:

```
anEmployee = (Person *)getEmployee();
```

но const_cast лучше по нескольким причинам. Во-первых, он очевиден и отвратителен. Он торчит в коде, как бревно в глазу, и это хорошо, потому что приведения опасны в любой форме. Писать их должно быть мучением, потому что их следует применять только в случае необходимости. Их должно быть легко найти, потому что приведения – это то, что проверяется в первую очередь при возникновении ошибок. Во-вторых, const_cast обладает меньшей силой, чем приведение в старом стиле, поскольку влияет только на квалификаторы типов. Это ограничение тоже имеет положительный эффект, потому что обеспечивает возможность точнее обозначить намерения. Применение приведения в старом стиле заставляет компилятор замолчать, потому что вы хотите, чтобы возвращаемым типом getEmployee был Person *. Применение const_cast заставляет компилятор замолчать, потому что вы хотите убрать const из возвращаемого типа getEmployee. Разница между этими двумя выражениями не так уж и велика (их даже объединяет их отвратительность), пока кто-нибудь не захочет изменить функцию getEmployee:

```
const Employee *getEmployee();          // значительное изменение!
```

«Правило кляпа», навязанное приведением в старом стиле, по-прежнему действует. Компилятор не просигнализирует о неверном преобразовании const Employee * в Person *, но пожалуется, если такое случится при использовании const_cast, потому что столь радикальное изменение выходит за рамки его возможностей. Короче говоря, const_cast предпочтительнее приведения в старом стиле, потому что const_cast еще более уродлив, его сложнее использовать и у него меньше возможностей.

Оператор static_cast применяется для переносимых на разные платформы приведений. Чаще всего он служит для приведения «вниз» по иерархии наследования от указателя или ссылки на базовый класс к указателю или ссылке на производный класс (см. также тему 27 «*Запросы возможностей*»):

```
Shape *sp = new Circle;
Circle *cp = static_cast<Circle *>(sp);  // нисходящее приведение
```

В данном случае static_cast обеспечивает безошибочную компиляцию, потому что sp действительно ссылается на объект Circle. Однако если бы sp

указывал на какой-нибудь другой тип Shape, при использовании cp мы, скорее всего, получили бы ошибку времени выполнения. Повторяю, эти новые операторы приведения безопаснее, чем старые, но не всегда безопасны.

Обратите внимание, что static_cast не должен изменять квалификаторы типа так, как это может const_cast[1]. Это означает, что порой необходимо использовать последовательность двух новых операторов приведения, чтобы достичь результата, обеспечиваемого приведением в старом стиле:

```
const Shape *getNextShape() { ... }
//...
Circle *cp =
   static_cast<Circle *>(const_cast<Shape *>(getNextShape()));
```

Стандарт не гарантирует поведения оператора reinterpret_cast, но, как правило, оно соответствует его имени: он работает с битами объекта и позволяет осуществлять преобразования между совершенно несвязанными типами:

```
hopeItWorks =                         // преобразуем целое в указатель
   reinterpret_cast<char *>(0x00ff0000);
int *hopeless =                       // преобразуем char * в int *
   reinterpret_cast<int *>(hopeItWorks);
```

Подобные вещи иногда приходится делать в коде низкого уровня, но такое преобразование не является переносимым. Применять его следует с осторожностью. Обратите внимание на разницу между reinterpret_cast и static_cast при нисходящем приведении от указателя на базовый класс к указателю на производный класс. Оператор reinterpret_cast обычно считает, что указатель на базовый класс является указателем на производный класс, и не меняет его значение, тогда как static_cast (и приведение в старом стиле, если на то пошло) осуществляет правильную работу с адресами (см. тему 28 «*Смысл сравнения указателей*»).

Разговор о приведении в рамках иерархии приводит нас к оператору dynamic_cast. Он обычно применяется для безопасного нисходящего приведения от указателя на базовый класс к указателю на производный класс (см., однако, тему 27 «*Запросы возможностей*»). От static_cast оператор dynamic_cast отличается тем, что нисходящее приведение может осуществляться только для полиморфного типа (т. е. тип приводимого выражения должен быть указателем на тип класса с виртуальной функцией) и правильность приведения проверяется во время выполнения. Однако безопас-

[1] Стандарт языка C++ позволяет добавлять квалификаторы const и volatile с помощью оператора static_cast, однако удаление указанных квалификаторов с помощью static_cast запрещается (см., например, ISO IEC 14882 п. 5.2.9).

ность не бесплатна. У оператора static_cast издержки обычно отсутствуют или минимальны, тогда как dynamic_cast предполагает существенные издержки времени выполнения.

```
const Circle *cp =
    dynamic_cast<const Circle *>( getNextShape() );
if( cp ) { ... }
```

Если getNextShape возвращает указатель на Circle (или какой-то наследуемый открыто от Circle класс, т. е. нечто, являющееся Circle; см. тему 2 «*Полиморфизм*»), приведение будет успешным и cp будет указывать на Circle. В противном случае cp будет нулевым. Обратите внимание, что объявление и тестирование можно комбинировать в одном выражении:

```
if( const Circle *cp
    = dynamic_cast<const Circle *>(getNextShape()) ) {...}
```

Это выгодно, потому что тем самым область видимости переменной cp ограничивается выражением if. Таким образом, cp, когда в ней исчезнет необходимость, просто уйдет из области видимости.

Несколько реже dynamic_cast применяется для нисходящего приведения к ссылочному типу:

```
const Circle &rc = dynamic_cast<const Circle &>(*getNextShape());
```

Операция аналогична приведению указателя с помощью dynamic_cast, но если приведение не удается, оператор формирует исключение типа std::bad_cast вместо возврата нулевого указателя. (Вспомните, что нулевых ссылок нет; см. тему 5 «*Ссылки – это псевдонимы, а не указатели*».) Можно сказать, что применение dynamic_cast к указателю – это вопрос («Этот указатель на Shape фактически указывает на Circle? Если нет, я могу заняться этим.»), тогда как применение dynamic_cast к ссылке является утверждением («Предполагается, что этот Shape есть не что иное, как Circle. Если это не так, значит, что-то не в порядке!»).

Как и со всеми остальными новыми операторами приведения, необходимость в dynamic_cast возникает лишь время от времени. Но из-за репутации «безопасного» приведения им часто злоупотребляют. Пример такого злоупотребления можно найти в теме 30 «*Фабричный метод*».

Тема 10 | Смысл константной функции-члена

С технической точки зрения константные функции-члены тривиальны. Однако их взаимоотношения с окружением могут быть сложными.

Типом указателя this в неконстантной функции-члене класса X является X *const. Другими словами, это константный указатель на неконстантный X (см. тему 7 «*Константные указатели и указатели на константу*»). Поскольку объект, на который ссылается this, неконстантный, он может быть изменен. Типом this в константной функции-члене класса X является const X *const — константный указатель на константный X. Поскольку объект, на который ссылается this, константный, он не может быть изменен. В этом разница между константными и неконстантными функциями-членами.

Вот почему можно изменять логическое состояние объекта с помощью константной функции-члена, даже если физическое состояние объекта остается неизменным. Рассмотрим следующую банальную реализацию класса X, в котором фигурирует указатель на буфер, выделенный для хранения некоторой части его состояния:

```
class X {
  public:
    X() : buffer_(0), isComputed_(false) {}
    //...
    void setBuffer() {
      int *tmp = new int[MAX];
      delete [] buffer_;
      buffer_ = tmp;
    }
```

```
      void modifyBuffer( int index, int value ) const  // противоречит нормам!
          { buffer_[index] = value; }
      int getValue() const {
          if( !isComputed_ ) {
             computedValue_ = expensiveOperation();   // ошибка!
             isComputed_ = true;                      // ошибка!
          }
          return computedValue_;
      }
    private:
      static int expensiveOperation();
      int *buffer_;
      bool isComputed_;
      int computedValue_;
};
```

Функция-член setBuffer должна быть неконстантной, потому что меняет поля своего объекта X. Однако modifyBuffer вполне может быть константной, потому что меняет не объект X, а лишь некоторые данные, на которые ссылается его поле buffer_member.

Это законно, но так поступать нельзя. Как недобросовестный адвокат, следующий букве закона, но при этом нарушающий его суть, программист на C++, пишущий константную функцию-член, которая изменяет логическое состояние своего объекта, будет как минимум осужден своими коллегами (если не компилятором). Это просто неправильно.

Наоборот, иногда функция-член, фактически объявленная константной, должна менять свой объект. Это обычная ситуация при нахождении значения путем «отложенного вычисления», когда для повышения производительности значение не вычисляется до первого запроса. В функции X::getValue делается попытка осуществить отложенное вычисление ресурсоемкого выражения. Но поскольку оно объявлено как константная функция-член, X::getValue не может задавать значения атрибутов isComputed_ и computedValue_ своего объекта X. В подобных ситуациях есть соблазн переступить закон и провести приведение, чтобы получить выгоду от возможности объявить функцию-член константной:

```
int getValue() const {
   if( !isComputed_ ) {
       X *const aThis = const_cast<X *const>(this);    // плохая идея!
       aThis->computedValue_ = expensiveOperation();
       aThis->isComputed_ = true;
   }
   return computedValue_;
}
```

Не поддавайтесь искушению. Чтобы справиться с этой ситуацией, надо объявить соответствующие данные-члены с модификатором mutable:

```
class X {
   public:
      //...
      int getValue() const {
         if( !isComputed_ ) {
            computedValue_ = expensiveOperation(); // все в порядке...
            isComputed_ = true;                    // тоже все в порядке...
         }
         return computedValue_;
      }
   private:
      //...
      mutable bool isComputed_;      // теперь может быть изменен
      mutable int computedValue_;    // теперь может быть изменен
};
```

Нестатические данные-члены класса могут быть объявлены с модификатором mutable, что позволит изменять их значения константным функциям-членам класса (как и неконстантным функциям-членам). Это в свою очередь обеспечивает возможность объявлять константной «логически константную» функцию-член, даже несмотря на то, что реализация требует изменения ее объекта.

Влиянием константности на тип указателя this функции-члена также объясняется то, как разрешение перегрузки функции может различать константные и неконстантные версии функции-члена. Рассмотрим следующий вездесущий пример перегруженного индексного оператора:

```
class X {
   public:
      //...
      int &operator [](int index);
      const int &operator [](int index) const;
      //...
};
```

Вспомним, что левый аргумент бинарного перегруженного оператора-члена передается как указатель this. Следовательно, при индексации объекта X его адрес передается как указатель this:

```
int i = 12;
X a;
a[7] = i;    // это X *const, потому что a не константа
const X b;
i = b[i];    // это const X *const, потому что b константа
```

Разрешение перегрузки сопоставит адрес константного объекта с указателем this, который указывает на константу. В качестве другого примера рассмотрим следующий бинарный оператор-нечлен с двумя константными аргументами:

```
X operator +( const X &, const X & );
```

Если понадобится объявить функцию-член, аналогичную этому перегруженному оператору, для сохранения константности левого аргумента он должен быть объявлен константной функцией-членом:

```
class X {
   public:
      //...
      X operator +( const X &rightArg );      // левый аргумент не константа!
      X operator +( const X &rightArg ) const; // левый аргумент - константа
      //...
};
```

Правильное программирование на C++ с применением констант не отличается технической сложностью, но подвергает испытанию дух программиста, в чем просматривается аналогия с повседневной жизнью.

Тема 11 | Компилятор дополняет классы

Программисты на C привыкли знать все о внутреннем устройстве и компоновке структур и писать код, зависящий от определенной компоновки. Java-программисты привыкли программировать, не зная о компоновке структуры своих объектов, и иногда думают, что при переходе на C++ эпоха неведения заканчивается. Однако написание безопасного и переносимого кода на C++ требует частичного признания непознаваемости структуры и компоновки объектов класса.

Для класса не всегда действует правило «что вижу, то и имею». Например, большинство программистов на C++ знают, что если класс объявляет одну или более виртуальных функций, компилятор вставит в каждый объект этого класса указатель на таблицу виртуальных функций. (На самом деле это не гарантируется стандартом, но все существующие компиляторы C++ реализуют виртуальные функции именно так.) Однако программисты на C++, находясь на этой опасной грани между знанием и опытом, часто пишут код исходя из предположения, что положение указателя таблицы виртуальных функций остается неизменным от платформы к платформе. Это грубейшая ошибка! Одни компиляторы помещают указатель в начале объекта, другие – в конце, а если задействовано множественное наследование, по объекту может быть разбросано несколько указателей таблицы виртуальных функций. Какие-либо предположения недопустимы.

И это еще не все. При виртуальном наследовании объекты могут отслеживать местоположение своих виртуальных подобъектов базового класса с помощью встроенных указателей, встроенных смещений или невстроенной информации. Указатель таблицы виртуальных функций может появиться, даже если в классе нет виртуальных функций! Разве я не говорил, что компилятор также может переупорядочивать члены данных не-

сколькими способами независимо от порядка, в котором они были объявлены? Настанет ли конец этому безумию?

Настанет. Для того чтобы получить класс, гарантированно аналогичный C-структуре, можно определить POD (plain old data – простые данные в стиле C). Конечно, к POD относятся не только встроенные типы, такие как `int`, `double` и аналогичные, но и типы, объявленные в C, подобные `struct` или `union`.

```
struct S {          // POD-структура
   int a;
   double b;
};
```

Работа с такими POD-структурами так же безопасна, как с соответствующими конструкциями C (степень этой безопасности в C++ не вызывает сомнений настолько же, как и в C). Однако если POD-структуры планируется задействовать в программном коде низкого уровня, то они должны и с точки зрения обработки их компилятором оставаться данными в стиле C, иначе все пропало:

```
struct S {          // больше не POD-структура!
      int a;
      double b;
   private:
      std::string c; // выполняется некоторая обработка компилятором,
                     // не характерная для POD
};
```

Но какое значение имеет это вмешательство компилятора для последующей работы с объектами класса, если разработчик не желает иметь дело только с POD? Оно означает, что работать с объектами класса можно только на высоком уровне, а не как с наборами битов. Высокоуровневая операция от платформы к платформе может делать «одно и то же», но выполняться будет по-разному.

Например, если необходимо скопировать объект класса, никогда нельзя использовать блочное копирование, обеспечиваемое стандартной функцией `memcpy` или самостоятельно разработанным ее эквивалентом. Они предназначены для копирования хранилищ, но не объектов (это отличие обсуждается в теме 35 «*Синтаксис размещения new*»). Необходимо использовать реализуемые объектом операции инициализации или присваивания. Конструктор объекта – это то место, где компилятор вызывает скрытый механизм, реализующий виртуальные функции объекта и т. п. Простое внедрение набора битов в неинициализированное хранилище может не привести к желаемому результату. Аналогично при копировании одного объекта в другой необходимо позаботиться о том, чтобы при этом

не произошла перезапись механизмов этого класса. Например, присваивание никогда не меняет значения указателей таблицы виртуальных функций объекта. Они задаются конструктором и не меняются в течение жизни объекта. Внедрение битов может разрушить эту нежную внутреннюю структуру. (См. также тему 13 «*Операции копирования*».)

Другая широко распространенная проблема связана с предположением о неизменном соответствии определенного члена класса заданному смещению в рамках объекта. Скажем, в каком-нибудь заумном коде нередко принимают, что нулевому смещению соответствует или указатель на таблицу виртуальных функций, или поле, являющееся первым в объявлении класса. Оба предположения правильны не более чем наполовину, и конечно же, они не могут быть истинными одновременно.

```
struct T {           // не POD
   int a_;           // смещение a_ неизвестно
   virtual void f(); // смещение vptr неизвестно
};
```

Я не собираюсь развивать данную тему, потому что этот список окажется довольно длинным, скучным и, наверное, неинтересным. Но в следующий раз, когда вы будете делать предположения о внутренней структуре своих классов, остановитесь, подумайте и избавьтесь от предрассудков!

Тема 12 | Присваивание и инициализация – это не одно и то же

Инициализация и присваивание – это разные операции, которые применяются и реализуются по-разному.

Давайте разложим все по полочкам. Присваивание имеет место, когда что-то присваивается. Все остальные операции копирования – это инициализация, включая инициализацию при объявлении, возвращении значения функцией, передаче аргумента и перехвате исключений.

Присваивание и инициализация – абсолютно разные операции не только потому, что используются в разных контекстах, но и потому, что выполняют абсолютно разные действия. Эта разница не столь очевидна для встроенных типов, таких как `int` или `double`, поскольку в их случае и присваивание, и инициализация состоят в простом копировании битов (но см. также тему 5 «*Ссылки – это псевдонимы, а не указатели*»):

```
int a = 12;  // инициализация, копирование 0X000C в a
a = 12;      // присваивание, копирование 0X000C в a
```

Однако для пользовательских типов ситуация может быть совершенно другой. Рассмотрим следующий простой нестандартный класс для работы со строками:

```
class String {
   public:
      String( const char *init ); // намеренно без ключевого слова «explicit»
      ~String();
      String( const String &that );
      String &operator =( const String &that );
      String &operator =( const char *str );
      void swap( String &that );
```

```
      friend const String                    // конкатенация
         operator +( const String &, const String & );
      friend bool operator <( const String &, const String & );
      //...
   private:
      String( const char *, const char * );  // вычислительный
      char *s_;
};
```

Инициализация объекта строкой символов проста. Выделяется буфер, достаточно большой для размещения копии строки символов, и затем происходит копирование.

```
String::String( const char *init ) {
   if( !init ) init = "";
   s_ = new char[ strlen(init)+1 ];
   strcpy( s_, init );
}
```

Деструктор делает то, что должен делать:

```
String::~String() { delete [] s_; }
```

Присваивание – несколько более сложная задача, чем создание:

```
String &String::operator =( const char *str ) {
   if( !str ) str = "";
   char *tmp = strcpy( new char[ strlen(str)+1 ], str );
   delete [] s_;
   s_ = tmp;
   return *this;
}
```

Присваивание похоже на уничтожение с последующим созданием. Для составных пользовательских типов цель (левая половина или `this`) должна быть очищена перед повторной инициализацией источником (правая половина или `str`). В нашем случае с типом `String` его существующий буфер должен быть высвобожден перед прикреплением нового буфера символов. В теме 39 «*Надежные функции*» объясняется порядок инструкций. (Между прочим, практически каждую неделю кому-нибудь да приходит в голову светлая мысль реализовать присваивание, явно вызывая деструктор, а для вызова конструктора прибегая к ключевому слову `new`. Это не всегда работает и небезопасно. Не делайте так.)

Корректная операция присваивания очищает левый аргумент, поэтому не допускается пользовательское присваивание для неинициализированного хранилища:

```
String *names = static_cast<String *>(::operator new( BUFSIZ ));
```

```
names[0] = "Sakamoto";  // ой! применение delete []
                        // к неинициализированному указателю!
```

В данном случае names (имена) ссылается на неинициализированное хранилище, потому что operator new вызван напрямую, что исключает неявную инициализацию стандартным конструктором String. Переменная names ссылается на область памяти, заполненную случайными битами. Когда во второй строке будет вызван оператор присваивания String, он попытается удалить массив по неинициализированному указателю. (Безопасный способ осуществления операции, подобной такому присваиванию, приведен в теме 35 «*Синтаксис размещения new*».)

У конструктора меньше работы, чем у оператора присваивания (т. к. конструктор предполагает, что хранилище неинициализировано), поэтому реализация для повышения эффективности иногда основывается на так называемом *вычислительном конструкторе* (*computational constructor*):

```
const String operator +( const String &a, const String &b )
    { return String( a.s_, b.s_ ); }
```

Двухаргументный вычислительный конструктор не рассматривается как часть интерфейса класса String, поэтому объявляется закрытым.

```
String::String( const char *a, const char *b ) {
   s_ = new char[ strlen(a)+strlen(b)+1 ];
   strcat( strcpy( s_, a ), b );
}
```

Тема 13 | Операции копирования

Копирующее создание и копирующее присваивание – это разные операции. С технической точки зрения у них нет ничего общего, но они близки друг другу «социально» и должны быть совместимыми.

```
class Impl;
class Handle {
   public:
      //...
      Handle( const Handle & );              // копирующий конструктор
      Handle &operator =( const Handle & );  // копирующее присваивание
      void swap( Handle & );
      //...
   private:
      Impl *impl_;   // указатель на реализацию Handle
};
```

Копирование встречается настолько часто, что соблюдать соглашения при его осуществлении еще важнее, чем обычно. Эти операции всегда объявляются попарно соответственно приведенной выше сигнатуре (см., однако, темы 43 «*Указатель auto_ptr – штука странная*» и 32 «*Предотвращение копирования*»). То есть для класса `X` копирующий конструктор должен быть объявлен как `X(const X &)`, а оператор копирующего присваивания – как `X &operator =(const X &)`. Как правило, целесообразно, и широко применяется определение функции-члена `swap` в том случае, если реализация `swap` как функции-члена дает преимущество в производительности или безопасности, по сравнению с традиционной `swap` (нечленом). Реализация обычной `swap`, не являющейся членом, проста:

```
template <typename T>
void swap( T &a, T &b ) {
```

```
    T temp(a);  // копирующий конструктор T
    a = b;      // копирующее присваивание T
    b = temp;   // копирующее присваивание T
}
```

Эта функция swap (идентичная swap стандартной библиотеки) определяется на основании операций копирования типа T и хороша, если реализация T небольшая и простая. В противном случае она может приводить к накладным расходам. Для класса, подобного Handle, есть вариант получше: просто поменять местами указатели на его реализацию.

```
inline void Handle::swap( Handle &that )
    { std::swap( impl_, that.impl_ ); }
```

Помните комедийный номер, в котором рассказывалось, как получить миллион долларов и не платить с них налоги? Во-первых, получаем миллион долларов... Точно так же можно показать, как написать безопасную операцию копирующего присваивания. Сначала получаем безопасный копирующий конструктор и безопасную операцию swap. Остальное просто:

```
Handle &Handle::operator =( const Handle &that ) {
    Handle temp( that ); // безопасное копирующее создание
    swap( temp );        // безопасная swap
    return *this;        // предполагаем, что уничтожение temp
                         // не сформирует исключения
}
```

Эта техника особенно хороша для классов-«дескрипторов», т. е. классов, состоящих преимущественно или полностью из указателей на их реализации. Как мы видели в предыдущем примере, создание безопасных операций swap для таких классов не отличается сложностью и дает большой эффект.

Тонкость этой реализации копирующего присваивания в том, что его поведение и поведение копирующего создания должны быть совместимыми. Это разные операции, однако заведомо предполагается, что их результаты будут неотличимы. То есть если записано

```
Handle a = ...
Handle b;
b = a;          // присваиваем a к b
```

или

```
Handle a = ...
Handle b( a );  // инициализируем b значением a
```

получаемое в результате значение и будущее поведение b должны быть идентичны независимо от того, как именно это значение получено – посредством присваивания или инициализации.

Такая совместимость исключительно важна при использовании стандартных контейнеров, потому что их реализации часто подменяют копирующее создание копирующим присваиванием, предполагая, что эти операции обеспечивают идентичные результаты (см. тему 35 «*Синтаксис размещения new*»).

Вот еще одна, наверное, более распространенная реализация копирующего присваивания:

```
Handle &Handle::operator =( const Handle &that ) {
   if( this != &that ) {
      // осуществляется присваивание...
   }
   return *this;
}
```

Часто с целью обеспечения правильности необходимо, а иногда это и более эффективно, выполнить проверку на наличие самоприсваивания, т. е. убедиться, что адреса левой (`this`) и правой (`that`) частей присваивания разные.

Большинство программистов C++ хотя бы раз в течение карьеры экспериментируют с идеей реализации виртуального копирующего присваивания. Ничего незаконного в ней нет, но она несколько запутанна, и лучше оставить ее в покое. То ли дело клонирование (см. тему 29 «*Виртуальные конструкторы и Прототип*»).

Тема 14 | Указатели на функции

Можно объявить указатель на функцию конкретного типа.

```
void (*fp)(int);  // указатель на функцию
```

Обратите внимание на обязательные круглые скобки, показывающие, что fp — это указатель на функцию, которая возвращает void, а не void * (см. тему 17 «*Разбираемся с операторами объявления функций и массивов*»). Как и указатель на данные, указатель на функцию может быть нулевым или ссылаться на функцию соответствующего типа.

```
extern int f( int );
extern void g( long );
extern void h( int );
//...
fp = f;   // ошибка! &f типа int(*)(int), не void(*)(int)
fp = g;   //ошибка! &g типа void(*)(long), не void(*)(int)
fp = 0;   // OK, присваивается нуль
fp = h;   // OK, указывает на h
fp = &h;  // OK, явно принимает адрес
```

Заметьте, что необязательно явно извлекать адрес функции при инициализации или присваивании ее адреса указателю на функцию. Компилятор знает, что необходимо получить адрес функции, поэтому в данном случае оператор & может отсутствовать.

Точно так же можно не разыменовывать указатель на функцию, чтобы вызвать функцию, на которую он ссылается, потому что компилятор сделает это сам:

```
(*fp)(12);  // явное разыменование
fp(12);     // неявное разыменование, результат аналогичен
```

Обратите внимание, что не существует универсальных указателей, способных указывать на функции любого типа, как указатель `void *` может ссылаться на любой тип данных. Еще необходимо отметить, что адрес нестатической функции-члена – это не указатель, поэтому указать на нее с помощью указателя на функцию невозможно (см. тему 16 «*Указатели на функции-члены – это не указатели*»).

Традиционно указатели на функции служат для реализации обратных вызовов (более эффективные реализации обратных вызовов продемонстрированы в темах 18 «*Объекты-функции*» и 19 «*Команды и Голливуд*»). *Обратный вызов* – это потенциальное действие, которое задается на этапе инициализации, чтобы быть вызванным в ответ на будущее событие. Например, если необходимо отреагировать на пожар, лучше заранее спланировать, как действовать в этом случае:

```
extern void stopDropRoll();
inline void jumpIn() { ... }
//...
void (*fireAction)() = 0;
//...
if( !fatalist ) {  // если вам не безразлично, что вы горите...
                   // на всякий случай задайте соответствующее действие!
   if( nearWater )
      fireAction = jumpIn;
   else
      fireAction = stopDropRoll;
}
```

Определив порядок действий, в другой части кода можно указать, когда и в каком случае его выполнять, не заботясь о сути действия:

```
if( ftemp >= 451 ) {   // если возник пожар...
   if( fireAction )    // ...и необходимо выполнить действие...
       fireAction();    // ...выполнить его!
}
```

Допускается указатель на встраиваемую функцию. Однако вызов встраиваемой функции через указатель на функцию невозможен, потому что в общем случае компилятор неспособен во время компиляции точно определить, какая функция будет вызываться. В нашем предыдущем примере `fireAction` (действие при пожаре) может указывать на любую из двух функций (или ни на одну из них); таким образом, в точке вызова компилятору ничего не остается, как сгенерировать код для непрямого вызова невстраиваемой функции.

Можно получить и адрес перегруженной функции:

```
void jumpIn();
```

```
void jumpIn( bool canSwim );
//...
fireAction = jumpIn;
```

Тип указателя применяется для осуществления выбора функции из предлагаемых возможных. В данном случае `fireAction` имеет тип `void(*)()`, таким образом, выбирается первая функция `jumpIn` (запрыгнуть).

В стандартной библиотеке указатели на функции выступают в качестве обратных вызовов в нескольких случаях. Самый примечательный – стандартная функция `set_new_handler`. Она задает обратный вызов, который должен быть инициирован, если глобальная функция `operator new` не может выделить память по запросу.

```
void begForgiveness() {
   logError( "Sorry!" );
   throw std::bad_alloc();
}
//...
std::new_handler oldHandler =
   std::set_new_handler(begForgiveness);
```

Стандартное имя типа для `new_handler` – `typedef`:

```
typedef void (*new_handler)();
```

Следовательно, обратный вызов должен быть функцией, не принимающей аргументы и возвращающей `void`. Функция `set_new_handler` присваивает функцию обратного вызова своему аргументу и возвращает предыдущий обратный вызов. Отдельной функции для получения и задания нет. Получение текущей функции обратного вызова напоминает хождение по кругу:

```
std::new_handler current
    = std::set_new_handler( 0 );  // получаем...
std::set_new_handler( current );  // ...и возвращаем в исходное состояние!
```

Стандартные функции `set_terminate` и `set_unexpected` следуют такой же схеме комбинированного получения и задания обратного вызова.

Тема 15 | Указатели на члены класса – это не указатели

Слово «указатель» в описаниях указателей на члены класса неуместно. Они не содержат адресов и ведут себя не как указатели.

Синтаксис объявления указателя на член не так уж и плох (если вы успели привыкнуть к синтаксису объявления обычных указателей):

```
int *ip;        // указатель на int
int C::*pimC;   // указатель на член типа int класса C
```

Единственное отличие в том, что символ `*`, заменяется на `имя_класса::*`, то есть имеет место ссылка на член класса `имя_класса`. Во всем остальном синтаксис аналогичен обычному объявлению указателя.

```
void *   *   *const * weird1;
void *A::*B::*const * weird2;
```

`weird1` — это указатель типа «указатель на константный указатель на указатель на указатель на `void`». `weird2` — это указатель типа «указатель на константный указатель на член `B` на указатель на член `A`, который является указателем на `void`». (Это всего лишь пример, обычно такие сложные или нелепые описания не встречаются.)

Обычный указатель содержит адрес. При разыменовании указателя получаем объект, находящийся по этому адресу:

```
int a = 12;
ip = &a;
*ip = 0;
a = *ip;
```

Указатель на член, в отличие от обычного указателя, не ссылается на конкретную область памяти. Он ссылается на конкретный член класса,

но не на конкретный член конкретного объекта. Поэтому обычно указатель на член данных считают смещением. Это не обязательно, потому что стандарт C++ ничего не говорит о том, как должен быть реализован указатель на член данных, оговариваются только его синтаксис и поведение. Однако большинство компиляторов реализуют указатели на члены данных как целое, содержащее значение, равное смещению до указываемого поля плюс один. (Смещение инкрементируется для того, чтобы значением 0 представлять нулевой указатель на поле.) Смещение указывает, на сколько байтов от начала объекта смещен конкретный член.

```
class C {
   public:
      //...
      int a_;
};
int C::*pimC; // указатель на член C типа int
C aC;
C *pC = &aC;
pimC = &C::a_;
aC.*pimC = 0;
int b = pC->*pimC;
```

Присваивая pimC значение &C::a_, мы в сущности задаем в pimC смещение a_ в рамках C. Давайте разберемся. Если a_ не статический член, применение & в выражении &C::a_ возвращает нам не адрес, а смещение. Обратите внимание, что это смещение применяется к любому объекту типа C. То есть если член отстоит на 12 байт от начала в одном объекте C, то на те же 12 байт он будет отстоять и от начала любого другого объекта C.

Чтобы обратиться к члену данных, имея в распоряжении его смещение в рамках класса, нам необходим адрес объекта этого класса. Вот где появляются необычные на вид операторы .* и ->*. Записывая pC->*pimC, мы увеличиваем адрес, хранящийся в pC, на смещение, хранящееся в pimC, чтобы организовать доступ к соответствующему члену данных объекта C, на который ссылается pC. Записывая aC.*pimC, мы увеличиваем адрес aC на смещение, хранящееся в pimC, чтобы организовать доступ к соответствующему члену данных объекта C, на который ссылается pC.

Указатели на члены данных используются не так широко, как указатели на функции-члены, но они позволяют наглядно проиллюстрировать понятие контрвариантности.

Имеет место предопределенное преобразование указателя на производный класс в указатель на любой из его открытых базовых классов. Часто говорят, что между производным классом и его открытыми базовыми классами устанавливается отношение «является». Это отношение часто вытекает естественным образом из анализа предметной области (см. те-

му 2 «*Полиморфизм*»). Следовательно, можно утверждать, что, например, Circle (Круг) является Shape (Форма) посредством открытого наследования. И C++ поддерживает нас, предоставляя неявное преобразование Circle * в Shape *. Неявного преобразования Shape * в Circle * нет, потому что такое преобразование не имело бы смысла. Может существовать много разных типов Shape, и не все они являются Circle. (К тому же глупо звучит: «Форма является кругом».)

В случае с указателями на члены класса наблюдается обратная ситуация: есть неявное преобразование указателя на член базового класса в указатель на член открытого производного класса, но нет преобразования указателя на член производного класса в указатель на член любого из его базовых классов. Эта концепция контрвариантности кажется нелогичной, если только не вспомнить о том, что указатель на член данных не является указателем на объект. Это смещение *в* объекте.

```
class Shape {
   //...
   Point center_;
   //...
};
class Circle : public Shape {
   //...
   double radius_;
   //...
};
```

Circle представляет собой Shape, таким образом, объект Shape содержит подобъект Circle. Следовательно, любое смещение в рамках Shape действительно также и в Circle.

```
Point Circle::*loc = &Shape::center_;  // OK, базовый к производному
```

Однако Shape необязательно представляет собой Circle, таким образом, смещение члена Circle не всегда действительно в рамках Shape.

```
double Shape::*extent =
   &Circle::radius_;                   // ошибка! производный к базовому
```

Нелишним будет отметить, что в Circle есть все члены данных его базового класса Shape (т. е. он наследует эти члены от Shape). C++ поддерживает неявное преобразование указателя на член Shape в указатель на член Circle. Нельзя сказать, что Shape содержит все члены данных Circle (Shape ничего не наследует от Circle). И C++ напоминает об этом, запрещая преобразование указателя на член Circle в указатель на член Shape.

Тема 16 | Указатели на функции-члены – это не указатели

Принимая адрес нестатической функции-члена, вы получаете не адрес, а указатель на функцию-член.

```
class Shape {
   public:
      //...
      void moveTo( Point newLocation );
      bool validate() const;
      virtual bool draw() const = 0;
      //...
};
class Circle : public Shape {
      //...
      bool draw() const;
      //...
};
//...
void (Shape::*mf1)( Point ) = &Shape::moveTo; // не указатель
```

Синтаксис объявления указателя на функцию-член на самом деле не сложнее, чем синтаксис объявления указателя на обычную функцию (который, надо сказать, не очень удачен; см. тему 17 «*Разбираемся с операторами объявления функций и массивов*»). Как и для указателей на данные-члены, необходимо использовать не `*`, а `имякласса`, тем самым показывая, что функция, на которую ссылаются, является членом `имякласса`. Однако, в отличие от обычного указателя на функцию, указатель на функцию-член может ссылаться на константную функцию-член:

```
bool (Shape::*mf2)() const = &Shape::validate;
```

Как и с указателем на данные-член, чтобы разыменовать указатель на функцию-член, необходим объект или указатель на объект. В случае с указателем на данные-член, чтобы организовать доступ к члену, к его смещению (содержащемуся в указателе на данные-член) необходимо добавить адрес объекта. В случае с указателем на функцию-член необходим адрес объекта, чтобы использовать его (или вычислить; см. тему 28 «*Смысл сравнения указателей*») как значение указателя this для вызова функции и, возможно, других целей.

```
Circle circ;
Shape *pShape = &circ;
(pShape->*mf2)();  // вызываем Shape::validate
(circ.*mf2)();     // вызываем Shape::validate
```

Операторы ->* и .* должны быть заключены в круглые скобки, потому что они имеют меньший приоритет, чем оператор (). И прежде чем вызывать функцию, необходимо выяснить, какую функцию вызывать! Это абсолютно аналогично использованию круглых скобок в выражении (a+b)*c, где мы хотим, чтобы сложение выполнялось раньше, чем умножение, имеющее больший приоритет.

Обратите внимание, что нет такого понятия, как «виртуальный указатель на функцию-член». Виртуальность – свойство самой функции-члена, а не указателя, ссылающегося на нее.

```
mf2 = &Shape::draw;  // draw виртуальная
(pShape->*mf2)();    // вызываем Circle::draw
```

Есть одна причина, по которой указатель на функцию-член не может быть реализован в общем случае как простой указатель на функцию. Реализация указателя на функцию-член должна сохранять в себе следующие сведения:

- является ли функция-член, на которую он ссылается, виртуальной или невиртуальной;
- где искать соответствующий указатель таблицы виртуальных функций (см. тему 11 «*Компилятор дополняет классы*»);
- какое смещение должно добавляться к указателю this функции или вычитаться из него (см. тему 28 «*Смысл сравнения указателей*»);
- возможно, другую информацию.

Указатель на функцию-член обычно реализуется как небольшая структура, содержащая всю эту информацию, хотя используются и многие другие реализации. Разыменование и вызов указателя на функцию-член обычно включает проверку хранящейся информации и в зависимости от условий выполнение соответствующей последовательности вызовов виртуальной или невиртуальной функции.

Как и указатели на данные-члены, указатели на функции-члены демонстрируют контрвариантность: существует предопределенное преобразование указателя на функцию-член базового класса в указатель на функцию-член производного класса, но не наоборот. Это имеет смысл, если предполагается, что функция-член базового класса посредством указателя this будет организовывать доступ только к членам базового класса, тогда как функция производного класса может делать попытки доступа к членам, не присутствующим в базовом классе.

```
class B {
   public:
      void bset( int val ) { bval_ = val; }
   private
      int bval_;
};
class D : public B {
   public:
      void dset( int val ) { dval_ = val; }
   private:
      int dval_;
};
B b;
D d;
void (B::*f1)(int) = &D::dset; // ошибка! производная функция в указателе
                               // на базовый класс
(b.*f1)(12);                   // ой! Доступ к несуществующему члену dval!
void (D::*f2)(int) = &B::bset; // OK, базовая функция в указателе
                               // на производный класс
(d.*f2)(11);                   // OK, записываем в унаследованное поле bval
```

Тема 17 | Разбираемся с операторами объявления функций и массивов

Основной причиной путаницы с объявлениями указателей на функции и указателей на массивы является более высокая приоритетность модификаторов функции и массива, чем модификатора указателя. Поэтому часто приходится использовать круглые скобки.

```
int *f1();      // функция, возвращающая int *
int (*fp1)();   // указатель на функцию, возвращающую int
```

Та же проблема существует и с модификатором массива, имеющим более высокий приоритет:

```
const int N = 12;
int *a1[N];      // массив из N элементов типа int *
int (*ap1)[N];   // указатель на массив, состоящий из N целых элементов
```

Конечно, если может существовать указатель на функцию или массив, то может существовать и указатель на этот указатель:

```
int (**ap2)[N];   // указатель на указатель на массив,
                  // состоящий из N элементов типа int
int *(*ap3)[N];   // указатель на массив, состоящий из N элементов типа int *
int (**const fp2)() = 0;  // константный указатель на указатель на функцию
int *(*fp3)();            // указатель на функцию, возвращающую int *
```

Заметьте, что тип функции или указателя на функцию зависит и от аргумента, и от возвращаемых типов.

```
char *(*fp4)(int,int);
char *(*fp5)(short,short) = 0;
fp4 = fp5;       // ошибка! несоответствие типов
```

Сложности появятся, если модификаторы функции и массива окажутся в одном объявлении. Рассмотрим следующую распространенную и неверную попытку объявить массив указателей на функцию:

```
int (*)()afp1[N];  // синтаксическая ошибка!
```

В приведенном выше (неправильном) объявлении модификатор функции () завершает объявление, а добавленное дальше имя afp1 обозначает начало синтаксической ошибки. Это аналогично такому объявлению массива:

```
int[N] a2;         // синтаксическая ошибка!
```

которое вполне работоспособно в Java, но недопустимо в C++. При правильном объявлении массива указателей на функцию объявляемое имя указывается там же, где и в простом указателе на функцию. Затем говорится, что необходимо создать массив этих указателей:

```
int (*afp2[N])();  // массив из N указателей на функцию,
                   // возвращающую элементы типа int
```

Выражения становятся слишком громоздкими, поэтому пора обратиться к typedef.

```
typedef int (*FP)(); // указатель на функцию возвращает элемент типа int
FP afp3[N];          // массив N элементов FP того же типа, что и afp2
```

Применение typedef для упрощения синтаксиса сложных объявлений — это знак заботы о тех несчастных, которые будут работать с кодом после вас. С typedef упрощается даже объявление стандартной функции set_new_handler:

```
typedef void (*new_handler)();
new_handler set_new_handler( new_handler );
```

Таким образом, new_handler (см. тему 14 «*Указатели на функции*») — это указатель на функцию, не принимающую аргументы и возвращающую void. set_new_handler — это функция, принимающая new_handler как аргумент и возвращающая new_handler. Все просто. Если сделать все это без typedef, ваша популярность среди тех, кому придется обслуживать ваш код, стремительно упадет:

```
void (*set_new_handler(void (*)()))();  // правильно, но жестоко
```

Также можно объявить ссылку на функцию.

```
int aFunc( double );           // функция
int (&rFunc)(double) = aFunc;  // ссылка на функцию
```

Ссылки на функции применяются редко и заполняют примерно ту же нишу, что и константные указатели на функции:

```
int (*const pFunc)(double) = aFunc; // константный указатель на функцию
```

Ссылки на массивы обеспечивают некоторую дополнительную возможность, не предоставляемую указателями на массивы, и обсуждаются в теме 6 «*Массив как тип формального параметра*».

Тема 18 | Объекты-функции

Часто нам требуется что-то, что действовало бы аналогично указателю на функцию. Но указатели на функции слишком неуклюжи, опасны и (давайте согласимся с этим) давно устарели. Как правило, их с успехом заменяют объекты-функции.

Объект-функция, как и умный указатель (см. тему 42 «*Умные указатели*»), – это обычный объект класса. Тогда как тип умного указателя перегружает операторы `->` и `*` (и, возможно, `->*`), чтобы сымитировать «прокачанный указатель» (pointer on steroids, улучшенный, более функциональный), тип объекта-функции перегружает оператор вызова функции `()`, чтобы создать «прокачанный указатель на функцию». Рассмотрим объект-функцию, вычисляющий при каждом вызове следующий элемент общеизвестного ряда Фибоначчи (1, 1, 2, 3, 5, 8, 13...):

```
class Fib {
   public:
      Fib() : a0_(1), a1_(1) {}
      int operator ()();
   private:
      int a0_, a1_;
};
int Fib::operator ()() {
      int temp = a0_;
      a0_ = a1_;
      a1_ = temp + a0_;
      return temp;
}
```

Объект-функция – это всего лишь обычный объект класса, но его член `operator ()` (или члены, если их больше одного) можно вызывать с помощью стандартного синтаксиса вызова функции.

```
Fib fib;
//...
cout << "next two in series: " << fib() << ' ';
cout << fib() << endl;
```

Компилятор распознает синтаксис fib() как вызов функции-члена operator() переменной fib. По смыслу этот вызов идентичен fib.operator(), но, вероятно, проще для восприятия. В данном случае преимущество объекта-функции перед функцией или указателем на функцию в том, что состояние, необходимое для вычисления следующего элемента ряда Фибоначчи, хранится в самом объекте Fib. Функции для сохранения состояния между вызовами пришлось бы прибегнуть к глобальным или локальным статическим переменным или какому-нибудь другому трюку. Возможно, информацию пришлось бы передавать в функцию явно. Также следует обратить внимание, что, в отличие от функции, использующей статические данные, в этом случае можно иметь одновременно несколько объектов Fib, осуществляющих вычисления независимо друг от друга.

```
int fibonacci () {
   static int a0 = 0, a1 = 1;  // проблематично...
   int temp = a0;
   a0 = a1;
   a1 = temp + a0;
   return temp;
}
```

Также есть весьма популярная возможность создать эффект указателя на виртуальную функцию, образуя иерархию объекта-функции с помощью виртуального оператора operator(). Рассмотрим вычисление интеграла методом аппроксимации площади под кривой, как показано на рис. 18.1.

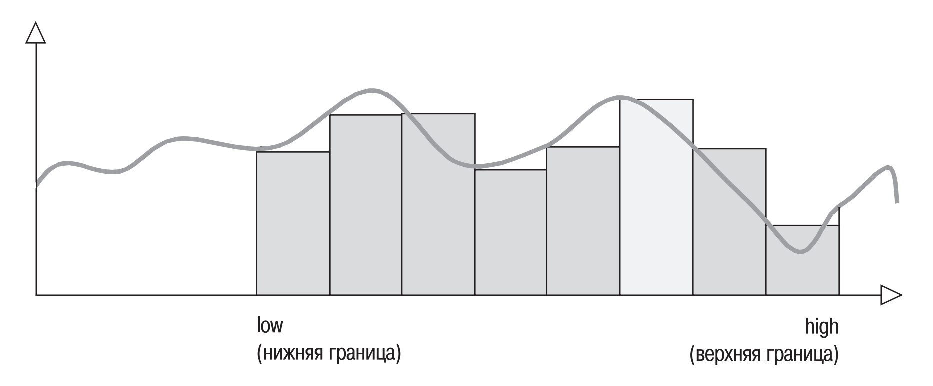


Рис. 18.1. *Численное интегрирование путем суммирования площадей прямоугольников (упрощенное)*

Для приближенного вычисления площади под кривой как суммы площадей прямоугольников (или реализации подобного механизма) функция интегрирования будет итеративно вызывать функцию для значений, находящихся между нижней и верхней границами.

```
typedef double (*F)( double );
double integrate( F f, double low, double high ) {
   const int numSteps = 8;
   double step = (high-low)/numSteps;
   double area = 0.0;
   while( low < high ) {
      area += f( low ) * step;
      low += step;
   }
   return area;
}
```

В этой версии передается указатель на функцию, для которой предполагается осуществить интегрирование.

```
double aFunc( double x ) { ... }
//...
double area = integrate( aFunc, 0.0, 2.71828 );
```

Такой вариант работоспособен, но негибок, потому что интегрируемая функция обозначается при помощи указателя на функцию. Этот код не может оперировать функциями, которым необходимы состояние или указатели на функции-члены. Альтернатива – создание иерархии объекта-функции. Базовым классом иерархии будет простой интерфейсный класс, объявляющий чистую виртуальную функцию `operator()`.

```
class Func {
   public:
      virtual ~Func();
      virtual double operator ()( double ) = 0;
};
double integrate( Func &f, double low, double high );
```

Теперь функция `integrate` способна интегрировать объект-функцию любого типа, который является `Func` (см. тему 2 «*Полиморфизм*»). Также интересно отметить, что тело `integrate` вообще не надо менять (хотя требуется повторная компиляция), потому что синтаксис вызова объекта-функции тот же, что и для указателя на функцию. Например, можно создать производный от `Func` тип, способный обрабатывать функции, не являющиеся членами:

```
class NMFunc : public Func {
   public:
```

```
      NMFunc( double (*f)( double ) ) : f_(f) {}
      double operator ()( double d ) { return f_( d ); }
   private:
      double (*f_)( double );
};
```

Это позволяет интегрировать все функции исходной версии:

```
double aFunc( double x ) { ... }
//...
NMFunc g( aFunc );
double area = integrate( g, 0.0, 2.71828 );
```

Также функции-члены можно интегрировать, заключая указатель на функцию-член и объект класса в соответствующий интерфейс-обертку (см. тему 16 «*Указатели на функции-члены – это не указатели*»):

```
template <class C>
class MFunc : public Func {
   public:
      MFunc( C &obj, double (C::*f)(double) )
         : obj_(obj), f_(f) {}
      double operator ()( double d )
         { return (obj_.*f_)( d ); }
   private:
      C &obj_;
      double (C::*f_)( double );
};
//...
AClass anObj;
MFunc<AClass> f( anObj, &AClass::aFunc );
double area = integrate( f, 0.0, 2.71828 );
```

Тема 19 | Команды и Голливуд

Если объект-функция выступает в качестве функции обратного вызова, это экземпляр паттерна Команда (Command).

Что такое обратный вызов? Предположим, вы отправляетесь в длительное путешествие и я одалживаю вам свою машину. Зная ее состояние, я, наверное, также дам вам запечатанный конверт с номером телефона и порекомендую позвонить по нему, если возникнут проблемы с двигателем. Это обратный вызов. Вам не надо знать номер телефона заранее (это может быть телефон хорошей станции техобслуживания, автобусной линии или городской свалки), и, кстати, он может вообще не понадобиться. В сущности задача по разрешению «проблемы с двигателем» распределена между вами (или «каркасом») и мной (или «клиентом каркаса»). Вы знаете, когда следует сделать что-то, но не знаете, что делать. Я знаю, что делать в случае конкретного события, но не знаю, когда делать. Вместе мы образуем единое целое.

Обратные вызовы – это распространенная техника программирования, которая обычно реализуется через простые указатели на функции (см. тему 14 «*Указатели на функции*»). Рассмотрим интерактивный тип `Button`, отображающий на экране кнопку с надписью и выполняющий действие при щелчке по ней.

```
class Button {
   public:
      Button( const string &label )
         : label_(label), action_(0) {}
      ~Button {... delete action_; ...}
      void setAction( void (*newAction)() )
         { action_ = newAction; }
      void onClick() const
```

```
        { if( action_ ) action_(); }
  private:
     string label_;
     void (*action_)();
     //...
};
```

Пользователь Button задает функцию обратного вызова и затем передает управление Button коду каркаса, который может определять момент нажатия кнопки и выполнять действие.

```
extern void playMusic();
//...
Button *b = new Button( "Anoko no namaewa" );
b->setAction( playMusic );
registerButtonWithFramework( b );
```

Такое разделение обязанностей часто называют *принципом Голливуда*: «Не звоните нам, мы сами вам позвоним». Мы настраиваем кнопку на выполнение определенного действия в случае нажатия. Код каркаса знает, что это действие должно быть инициировано при нажатии кнопки.

Однако применение простого указателя на функцию в качестве функции обратного вызова имеет строгие ограничения. Функции часто обрабатывают некоторые данные, но с указателем на функцию не ассоциированы никакие данные. Откуда функция playMusic (воспроизвести музыку) из предыдущего примера может знать, какую песню воспроизводить? Чтобы быстро решить эту задачу, обычно либо строго ограничивают область применения функции

```
extern void playAnokoNoNamaewa();
//...
b->setAction( playAnokoNoNamaewa );
```

либо прибегают к сомнительной и опасной практике применения глобальной переменной:

```
extern const MP3 *theCurrentSong = 0;
//...
const MP3 anokoNoNamaewa ( "AnokoNoNamaewa.mp3" );
theCurrentSong = &anokoNoNamaewa;
b->setAction( playMusic );
```

Как правило, вместо указателя на функцию лучше применять объект-функцию. Объект-функция – или чаще иерархия объектов-функций – в сочетании с принципом Голливуда образует экземпляр паттерна Команда.

Очевидное преимущество объектно-ориентированного подхода состоит в том, что объект-функция может инкапсулировать данные. Другой положительный момент – объект-функция может обладать динамическим поведением, посредством виртуальных членов. То есть возможна иерархия взаимосвязанных объектов-функций (см. тему 18 «*Объекты-функции*»). Есть и третий плюс, но он будет рассмотрен позже. А пока доработаем `Button` с применением паттерна Команда:

```
class Action {                              // паттерн Команда
   public:
      virtual ~Action();
      virtual void operator ()() = 0;
      virtual Action *clone() const = 0;  // Прототип
};
class Button {
   public:
      Button( const std::string &label )
          : label_(label), action_(0) {}
      ~Button {... delete action_; ...}
      void setAction( const Action *newAction ) {
         Action *temp = newAction->clone();
         delete action_;
         action_ = temp;
   }
   void onClick() const
         { if( action_ ) (*action_)(); }
   private:
      std::string label_;
      Action *action_;                      // Команда
      //...
};
```

Теперь `Button` может работать с любым объектом-функцией, представляющим собой `Action`, например таким:

```
class PlayMusic : public Action {
   public:
      PlayMusic( const std::string &songFile )
          : song_(songFile) {}
      void operator ()();                   // воспроизводит песню
      //...
   private:
      MP3 song_;
};
```

Инкапсулированные данные (в данном случае воспроизводимая песня) обеспечивают и гибкость, и безопасность объекта-функции `PlayMusic`.

```
Button *b = new Button ("Anoko no namaewa");
b->setAction (&PlayMusic ("AnokoNoNamaewa.mp3"));[1]
```

Так что же это за таинственное третье преимущество паттерна Команда, о котором упоминалось ранее? Работать с иерархией классов удобнее, чем с более примитивной и менее гибкой структурой указателя на функцию. Наличие иерархии паттерна Команда позволило сочетать паттерн Прототип (Prototype) с паттерном Команда, для того чтобы создавать клонируемые команды (см. тему 29 «*Виртуальные конструкторы и Прототип*»). Можно продолжать в том же духе и, комбинируя паттерны Команда и Прототип, создавать новые паттерны, обеспечивая тем самым дополнительную гибкость.

[1] Возможно, не все компиляторы обрадуются получению ссылки на временный объект. Для этого случая есть альтернативный код:

```
PlayMusic pm ("AnokoNoNamaewa.mp3");
b->setAction (&pm);
```

Тема 20 | Объекты-функции STL

Как вообще можно было обходиться без STL? Она не только упрощает и ускоряет написание сложного кода, но и позволяет получить стандартизованный и высоко оптимизированный код.

```
std::vector<std::string> names;
//...
std::sort( names.begin(), names.end() );
```

Еще одно замечательное свойство библиотеки STL в том, что она легко настраивается. В приведенном выше коде для сортировки вектора строк применялся оператор класса `string` «меньше чем». Но не всегда этот оператор есть в распоряжении или, возможно, не всегда требуется сортировка по возрастанию.

```
class State {
   public:
      //...
      int population() const;
      float aveTempF() const;
      //...
};
```

В классе `State` (который представляет штат государства) нет оператора «меньше чем». Наверное, и необходимости его реализовывать нет, потому что непонятно, что значит для одного штата быть меньше другого (сравниваются имена, население, процентное соотношение депутатов, находящихся под следствием, и т. д.). К счастью, STL позволяет в подобных ситуациях определять альтернативную операцию, подобную «меньше чем». Такие операции называются *компараторами*, потому что в них сравниваются два значения:

```
inline bool popLess( const State &a, const State &b )
    { return a.population() < b.population(); }
```

Создав компаратор для State, можно использовать его для сортировки:

```
State states[50];
//...
std::sort( states, states+50, popLess );  // по населению
```

Здесь в качестве компаратора передается указатель на функцию popLess (вспомните, что при передаче в качестве аргумента имя функции *разлагается* в указатель на функцию; так же и имя массива states разлагается в указатель на его первый элемент). Поскольку popLess передается как указатель на функцию, она не будет встроена в sort, что нежелательно, если важна скорость сортировки (см. тему 14 «*Указатели на функции*»).

Ситуацию можно исправить, применив в качестве компаратора объект-функцию:

```
struct PopLess : public std::binary_function <State,State,bool> {
    bool operator ()( const State &a, const State &b ) const
        { return popLess( a, b ); }
};
```

Тип PopLess – это характерный пример правильно построенного объекта-функции STL. Во-первых, это объект-функция. Он перегружает оператор вызова функции, поэтому может быть вызван с помощью обычного синтаксиса вызова функции. Это важно, потому что универсальные алгоритмы STL, такие как sort, написаны таким образом, что для создания их экземпляра может использоваться как указатель на функцию, так и объект-функция, при условии, что к ним применим обычный синтаксис вызова функции. Объект-функция с перегруженным operator() удовлетворяет этому синтаксическому требованию.

Во-вторых, он является производным от стандартного базового класса binary_function. Этот механизм позволяет другим частям реализации STL обращаться к объекту-функции во время компиляции (см. тему 53 «*Встроенная информация о типе*»). В данном случае происхождение от binary_function обеспечивает возможность получать информацию о типе аргумента и возвращаемом типе объекта-функции. Хотя здесь эта возможность не задействуется, с уверенностью можно сказать, что ее будет использовать кто-то другой, ведь мы хотим, чтобы нашим типом PopLess пользовались остальные.

В-третьих, у объекта-функции нет данных-членов, нет виртуальных функций, нет прямо объявленных конструкторов или деструктора и реализация operator () встраиваемая. Выступающие в качестве компараторов STL объекты-функции должны быть компактными, простыми и быстрыми. Можно создавать объекты-функции STL с довольно большими реализациями, но в основном это нецелесообразно. Другая причина сведения к минимуму или отказа от использования данных-членов в объекте-функ-

ции, предназначаемой для работы с STL, состоит в том, что реализации STL могут создавать несколько копий объекта-функции и предполагать идентичность всех этих копий. Проще всего гарантировать идентичность всех копий обекта, если объект не имеет данных вообще.

Теперь можно осуществить сортировку с помощью объекта-функции:

```
sort( states, states+50, PopLess() );
```

Обратите внимание на круглые скобки, стоящие после `PopLess` в данном вызове. `PopLess` – это тип, но объект этого типа приходится передавать как аргумент функции. Добавляя круглые скобки после имени типа `PopLess`, мы создаем безымянный временный объект `PopLess`, существующий в течение вызова функции. (Эти безымянные объекты называют *анонимными временными объектами* (*anonytous temporary*); мне очень нравится этот колоритный термин.) Можно было бы объявить и передавать именованный объект:

```
PopLess comp;
sort( states, states+50, comp );
```

Однако удобнее просто передавать анонимный временный объект.

Положительный эффект применения объекта-функции в качестве компаратора в том, что сравнение будет подставлено в строку, тогда как в случае применения указателя на функцию этого сделать нельзя. Преимущество встраиваемого вызова заключается в следующем: при создании экземпляра шаблона функции компилятору известен тип компаратора (`PopLess`), что, в свою очередь, дает ему информацию о будущем вызове `PopLess::operator ()` и обеспечивает возможность встроить эту функцию, используя встраиваемый вложенный вызов `popLess`.

Еще один распространенный случай применения объектов-функций STL – использование в качестве предикатов. Предикат – это логическая операция в отношении одного объекта. (Компаратор можно рассматривать как разновидность бинарного предиката.)

```
struct IsWarm : public std::unary_function<State,bool> {
   bool operator ()( const State &a ) const
      { return a.aveTempF() > 60; }
};
```

Рекомендации по разработке предикатов STL аналогичны рекомендациям по компараторам STL, за исключением того, конечно, что предикаты STL представляют собой унарные, а не бинарные функции. Основываясь на полученных ранее результатах сортировки `State`, с помощью соответствующих предикатов нетрудно найти теплое неперенаселенное местечко:

```
State *warmandsparse = find_if( states, states+50, IsWarm() );
```

Тема 21 | Перегрузка и переопределение – это не одно и то же

У перегрузки и переопределения нет ничего общего. Ничего. Непонимание этой разницы или небрежное отношение к терминам привели к невообразимой путанице и стали причиной несметного количества ошибок.

Перегрузка имеет место, когда в одной области видимости есть две или более функции с одинаковыми именами и разными сигнатурами. Сигнатура функции состоит из количества и типа объявленных параметров (их еще называют *формальными* параметрами). Найдя в области видимости несколько функций с одинаковыми именами, компилятор выбирает из них лишь одну, формальные параметры которой больше всего соответствуют фактическим параметрам вызова функции (см. также темы 24 «*Поиск функции-члена*» и 25 «*Поиск, зависимый от типов аргументов*»). Вот это и есть перегрузка.

Переопределение имеет место, когда имя и сигнатура функции производного класса полностью совпадают с виртуальной функцией базового класса. В этом случае для виртуальных вызовов производного объекта реализация функции производного класса заместит функцию, унаследованную от базового класса. Переопределение меняет поведение класса, но не его интерфейс (но см. тему 31 «*Ковариантные возвращаемые типы*»).

Рассмотрим следующий простой базовый класс:

```
class B {
  public:
    //...
    virtual int f( int );
    void f( B * );
```

```
        //...
};
```

Имя f перегружено в B, потому что в одной области видимости находятся две функции с именем f. (Перегрузка выделена как плохой код по двум причинам. Вероятно, не стоит перегружать виртуальную функцию и, возможно, не следует перегружать функции и с целочисленным типом, и с типом указателя. Обратитесь к книгам [2] и [5] соответственно, чтобы понять почему.)

```
class D : public B {
    public:
        int f( int );
        int f( B * );
};
```

Функция-член D::f(int) переопределяет виртуальную функцию базового класса B::f(int). Функция-член D::f(B *) ничего не переопределяет, потому что B::f(B *) не является виртуальной. Однако она перегружает D::f(int). Обратите внимание, что эта функция не перегружает члены базового класса B::f, потому что они находятся в другой области видимости (см. тему 63 «*Необязательные ключевые слова*»).

Перегрузка и переопределение – разные концепции. Формальное понимание их разницы исключительно важно для тех, кто хочет глубоко разбираться в углубленных аспектах проектирования интерфейсов базовых классов.

Тема 22 | Шаблонный метод

Шаблонный метод (Template Method) не имеет ничего общего с шаблонами C++. Скорее это средство, с помощью которого проектировщик базового класса может дать четкие инструкции по реализации контракта базового класса проектировщикам производных классов (см. тему 2 «*Полиморфизм*»). Однако даже если вам кажется, что этот паттерн должен быть назван иначе, пожалуйста, придерживайтесь стандартного названия «Шаблонный метод». Одним из главных преимуществ использования паттернов является вводимый ими стандартный словарь технических терминов (см. тему 3 «*Паттерны проектирования*»).

Базовый класс определяет контракт с внешним миром преимущественно посредством своих открытых функций-членов. Дополнительные детали для производных классов описываются защищенными функциями-членами. Закрытые функции-члены тоже могут использоваться как часть реализации класса (см. тему 12 «*Присваивание и инициализация – это не одно и то же*»). Данные-члены должны быть закрытыми, поэтому оставим их за рамками данного обсуждения.

Решение по поводу того, какой должна быть функция-член базового класса – виртуальной, невиртуальной или чисто виртуальной, – вытекает главным образом из того, как будет реализовываться поведение этой функции в производном классе. Ведь коду, использующему интерфейс базового класса, на самом деле все равно, как объект реализует конкретную операцию. Он хочет осуществлять эту операцию, а уж объекту решать, как правильно реализовать ее.

Если функция-член базового класса невиртуальная, проектировщик базового класса определяет инвариант иерархии базового класса. Производные классы не должны скрывать невиртуальную функцию базового класса за своими членами с таким же именем (см. тему 24 «*Поиск функ-*

ции-члена»). Если контракт, определенный базовым классом, не подходит, необходимо искать другой базовый класс. Нельзя пытаться переписать этот контракт.

Виртуальные и чисто виртуальные функции определяют операции, реализации которых могут быть настроены производными классами посредством переопределения. Функция, не являющаяся чисто виртуальной, предоставляет реализацию по умолчанию и не требует переопределения, тогда как чисто виртуальная функция должна быть переопределена в конкретном (т. е. не абстрактном) производном классе. Оба вида виртуальной функции позволяют производному классу полностью заменять свою реализацию, сохраняя интерфейс.

Применение шаблонного метода обеспечивает проектировщику базового класса промежуточный уровень контроля между «используй ее или оставь ее в покое» для невиртуальной функции и «если не нравится, замени все» для виртуальной функции. Шаблонный метод фиксирует общую структуру реализации, но некоторую ее часть адресует производным классам. Обычно Шаблонный метод реализован как открытая невиртуальная функция, вызывающая защищенные виртуальные функции. Производный класс должен принимать общую реализацию, определенную в унаследованной невиртуальной функции базового класса, но может настроить ее поведение ограниченным числом способов, переопределяя защищенные виртуальные функции, которые она инициирует.

```
class App {
   public:
      virtual ~App();
      //...

      void startup() { // Шаблонный метод
         initialize();
         if( !validate() )
            altInit();
      }
   protected:
      virtual bool validate() const = 0;
      virtual void altInit();
      //...
   private:
      void initialize();
      //...
};
```

Невиртуальный Шаблонный метод `startup` вызывает функции, настроенные производными классами:

```
class MyApp : public App {
   public:
      //...
   private:
      bool validate() const;
      void altInit();
      //...
};
```

Шаблонный метод – это пример принципа Голливуда в действии: «Не звоните нам, мы сами вам позвоним» (см. тему 19 «*Команды и Голливуд*»). Поток выполнения функции `startup` определяется базовым классом. Функция `startup`, в свою очередь, вызывается клиентами интерфейса базового класса. Проектировщики производных классов не знают, когда будут вызваны функции `validate` или `altInit`. Но им известно, что должны делать `validate` и `altInit`, когда будут вызваны. Совместно базовый и производные классы обеспечивают полную реализацию функции.

Тема 23 | Пространства имен

Глобальная область видимости была переполнена. Все кому не лень реализовывали библиотеки, использующие одни и те же имена для разных классов и функций. Например, многие библиотеки желали включить класс с именем `String`, но если использовать две разные библиотеки, определяющие тип `String`, возникает ошибка многократного определения или что-нибудь похуже. Различные дополнительные подходы к решению этой проблемы (соглашения о присваивании имен, препроцессор и другие) только ухудшали ситуацию. На помощь пришли пространства имен.

В каком-то смысле пространства имен вводят дополнительную сложность (см. тему 25 «*Поиск, зависимый от типов аргументов*»), но в большинстве своем упрощают дело и работать с ними не очень трудно. Пространство имен – это подраздел глобальной области видимости:

```
namespace org_semantics {
    class String { ... };
    String operator +( const String &, const String & );
    // другие классы, функции, объявления типов и т.д...
}
```

В данном фрагменте кода пространство имен `org_semantics` открывается, объявляется несколько полезных функций и затем пространство имен закрывается фигурной скобкой. Всегда можно что-то добавить в пространство имен, просто повторив эту операцию. Пространства имен являются «открытыми».

Обратите внимание, что некоторые имена пространства имен `org_semantics` объявлены, но не описаны. Чтобы описать эти имена, можно повторно открыть пространство имен:

```
namespace org_semantics {
```

```
    String operator +( const String &a, String &b ) { // ой!
        //...
    }
}
```

В качестве альтернативы можно просто уточнить описание именем пространства имен без повторного открытия пространства имен:

```
org_semantics::String
org_semantics::operator +(
    const org_semantics::String &a,
    const org_semantics::String &b ) {
    //...
}
```

Преимущество такого подхода в том, что он не позволяет по неосторожности объявить новое имя пространства имен (как мы сделали, когда не включили `const` во втором аргументе первого определения `operator+`). Надо сказать, это бесконечное повторение `org_semantics` в данном случае может быть утомительным, но такова цена безопасности! Мы обсудим некоторые подходы, которые могут несколько улучшить ситуацию.

Если необходимо использовать имя, описанное в определенном пространстве имен, надо сообщить компилятору, в каком пространстве имен можно его найти:

```
org_semantics::String s( "Hello, World!" );
```

Хотя часть стандартной библиотеки C++ осталась в глобальном пространстве имен (на ум приходят стандартные глобальные операторы `operator new`, `operator delete`, `operator new[]` и `operator delete[]` для создания и уничтожения массивов), основной ее состав теперь находится в пространстве имен `std` (от standard – стандартное). В большинстве случаев использование стандартной библиотеки требует указания имени пространства имен `std`:

```
#include <iostream>
#include <vector>
//...
void aFunc() {
    vector<int> a;            // ошибка! Я не вижу вектора!
    std::vector<int> b;       // О, вот он!
    cout << "Oops!" << endl;  // ошибки!
    std::cout << "Better!" << std::endl; // OK
    //...
}
```

Очевидно, что необходимость постоянного явного указания имени пространства имен утомляет. Избежать этого позволяет директива using.

```
void aFunc() {
   using namespace std;       // директива using
   vector<int> a;             // OK
   cout << "Hello!" << endl;  // OK
   //...
}
```

Директива using («используется») по сути «импортирует» имена из пространства имен, делая их доступными без квалификации в области видимости директивы using. В данном случае директива using действует до конца тела функции, а затем необходимо опять возвращаться к явной квалификации. По этой причине многие программисты на C++ (даже те, кому «виднее») предлагают помещать директиву using в глобальную область видимости:

```
#include <iostream>
#include <vector>
using namespace std;
using namespace org_semantics;
```

Это неудачная идея. Получается, что мы вернулись практически к тому, с чего начали: все имена пространства имен доступны везде, сея путаницу и неразбериху. Это особенно нежелательно в заголовочном файле, поскольку неудачное решение распространяется на все файлы, включающие данный заголовочный файл. В заголовочных файлах, как правило, лучше явно указывать квалификаторы, а директивы using приберечь для меньших областей видимости (таких как тела функций или блоки в рамках функции), где их действие ограничено и легче контролируется. По сути, в заголовочных файлах необходимо строго следовать правилам, в файлах реализации – выполнять рекомендации, а внутри функции можно расслабиться.

Один любопытный аспект директив using – имена пространства имен становятся доступными, как будто они объявлены глобально, а не в той области видимости, в которой используется директива using. Локальные имена перекрывают имена пространства имен:

```
void aFunc() {
   using namespace std; // директива using
   //...
   int vector = 12;     // неудачное имя для локальной переменной...
   vector<int> a;       // ошибка! std::vector скрыт
   std::vector<int> b;  // OK, можно использовать явную квалификацию
   //...
```

```
}
```

В качестве альтернативы можно назвать объявление `using` (using declaration), обеспечивающее доступ к имени пространства имен через фактическое объявление:

```
void aFunc() {
    using std::vector;   // using-объявление
    //...
    int vector = 12;     // ошибка! Повторное объявление vector
    vector<int> a;       // OK
    //...
}
```

`Using`-объявления зачастую представляют собой золотую середину между утомительными явными квалификациями и избыточным применением директив `using`, особенно если если в разделе кода используется лишь пара имен из двух или более пространств, но зато неоднократно:

```
void aFunc() {
    using std::cout;
    using std::endl;
    using org_semantics::String;
    String a, b, c;
    //...
    cout << a << b << c << endl;
    // и т.д.
}
```

Другой способ борьбы с длинными утомительными именами пространств имен состоит в применении псевдонимов:

```
namespace S = org_semantics;
```

Теперь `S` может использоваться в области видимости псевдонима вместо `org_semantics`. Как и в случае директивы `using`, лучше избегать применения псевдонимов пространств имен в заголовочных файлах. (Кроме всего прочего, `S`, скорее всего, будет конфликтовать с другими именами намного чаще, чем `org_semantics`...)

Завершим краткий обзор пространств имен рассмотрением анонимных пространств имен:

```
namespace {
    int anInt = 12;
    int aFunc() { return anInt; }
}
```

Поведение этого анонимного пространства имен аналогично следующему, где __compiler_generated_name__ («сгенерированное компилятором имя») уникально для каждого анонимного пространства имен:

```
namespace __compiler_generated_name__ {
    int anInt = 12;
    int aFunc() { return anInt; }
}
using namespace __compiler_generated_name__;
```

Это самый передовой способ избежать объявления функций и переменных со статическим связыванием. В приведенном выше анонимном пространстве имен обе функции — anInt и aFunc — имеют внешнее связывание, но доступ к ним может быть осуществлен только в рамках единицы трансляции (т. е. в полученном после предварительной обработки файле), в которой они находятся, точно так же, как к статическим переменным.

Тема 24 | Поиск функции-члена

Вызов функции-члена включает три этапа. Сначала компилятор ищет имя функции. Затем из доступных кандидатов он выбирает наиболее подходящую функцию. Далее он проверяет, имеет ли вызывающий доступ к выбранной функции. Вот и все. Надо сказать, каждый из этих этапов (особенно первые два; см. темы 25 «*Поиск, зависимый от типов аргументов*» и 26 «*Поиск операторной функции*») может быть сложным, но механизм сопоставления функций в целом предельно прост.

Большинство ошибок, связанных с сопоставлением функций, происходит не от незнания сложного механизма поиска имени компилятором и алгоритмов сопоставления перегруженных функций, а от непонимания простой последовательной природы этапов сопоставления. Рассмотрим следующий фрагмент кода:

```
class B {
  public:
    //...
    void f( double );
};
class D : public B {
    void f( int );
};
//...
D d;
d.f( 12.3 ); // сбивает с толку
```

Какой член вызывается?

Шаг 1: Ведем поиск имени функции. Поскольку вызывается член объекта `D`, поиск начинается в области видимости `D`. Сразу же обнаруживается `D::f`.

Шаг 2: Из доступных кандидатов выбираем наиболее подходящую функцию. В данном случае имеется только один кандидат, D::f, поэтому пытаемся сопоставить его. Это можно сделать путем преобразования фактического аргумента 12.3 из double в int. (Это допустимо, но, как правило, нежелательно, поскольку теряется точность.)

Шаг 3: Проверяем возможность доступа. Может возникнуть ошибка, потому что D::f — закрытый.

Наличие более подходящей доступной функции в базовом классе не имеет значения, потому что компилятор, обнаружив функцию во внутренней области видимости, не продолжает поиск имени во внешних областях видимости. Имя из внутренней области видимости *скрывает* такое же имя во внешней области. Такой перегрузки, как в Java, не происходит.

На самом деле даже необязательно, чтобы имя было именем функции:

```
class E : public D {
   int f;
};
//...
E e;
e.f( 12 ); // ошибка!
```

В данном случае возникает ошибка компиляции, потому что поиск имени f в области видимости приводит к члену данных, а не функции-члену. Это, кстати, одна из основных причин, по которой следует установить и соблюдать простое соглашение о присваивании имен. Если бы член данных E::f имел имя f_ или m_f, он бы не скрыл унаследованную функцию базового класса f.

Тема 25 | Поиск, зависимый от типов аргументов

Пространства имен имеют огромное влияние на современные программы и проекты C++ (см. тему 23 «*Пространства имен*»). Отчасти это влияние очевидно, например в наличии объявлений и директив `using` и квалификаторов имен. Однако пространства имен оказывают и менее заметное с точки зрения синтаксиса, но не менее фундаментальное и важное влияние. Сюда относится поиск, зависимый от типов аргументов (Argument dependent lookup, ADL). Как и многие другие возможности C++, ADL потенциально сложен, но в обычной работе он прост и решает больше проблем, чем создает.

Идея, лежащая в основе ADL, проста. При поиске имени функции, указанного в выражении вызова функции, компилятор также будет проверять пространства имен, содержащие типы аргументов вызова функции. Рассмотрим следующий код:

```
namespace org_semantics {
   class X { ... };
   void f( const X & );
   void g( X * );
   X operator +( const X &, const X & );
   class String { ... };
   std::ostream operator <<( std::ostream &, const String & );
}
//...
int g( org_semantics::X * );
void aFunc() {
   org_semantics::X a;
   f( a );       // вызываем org_semantics::f
```

```
    g( &a );      // ошибка! неоднозначность...
    a = a + a;    // вызываем org_semantics::operator +
}
```

Обычный поиск не выявит функции org_semantics::f, потому что она вложена в пространство имен и перед f не указан квалификатор пространства имен. Однако тип аргумента a определен в пространстве имен org_semantics, поэтому компилятор ведет поиск подходящих функций и в нем.

Конечно, такое сложное правило, как ADL, может неоднократно заставить программиста «почесать затылок». Вызов g с указателем на org_semantics::X – именно такой случай. Здесь программист, наверное, полагает, что компилятор обнаружит глобальную g. Но поскольку фактический аргумент относится к типу org_semantics::X *, при поиске были включены g и из этого пространства имен, и вызов стал неоднозначным. Поразмыслив, можно сказать, что на самом деле эта неоднозначность не так страшна, потому что, скорее всего, программист намеревался вызвать функцию org_semantics::g, а не ::g. Выявив неоднозначность, программист может или устранить ее, или переименовать одну из функций.

Обратите внимание, что, хотя при вызове g в результате обнаруживаются две функции-кандидата на разрешение перегрузки, ::g не перегружает org_semantics::g, потому что они объявлены в разных областях видимости (см. тему 21 «*Перегрузка и переопределение – это не одно и то же*»). ADL – это особенность вызова функции, а перегрузка – особенность ее объявления.

Реальную пользу от применения ADL можно увидеть при работе с инфиксными вызовами перегруженных операторов, например при использовании operator + в aFunc. Здесь инфиксное выражение a+a эквивалентно вызову operator +(a,a), и ADL обнаружит перегруженный operator+ в пространстве имен org_semantics (см. также тему 26 «*Поиск операторной функции*»).

На самом деле большинство программистов на C++ активно применяют ADL, даже не осознавая этого. Рассмотрим следующее обычное применение <iostream>:

```
org_semantics::String name( "Qwan" ); // QWAN - QualityWithoutAName, качество без
std::cout << "Hello, " << name;       // без названия, не поддающееся формулировке
```

В данном случае первый (самый левый) вызов operator<<, по всей вероятности, представляет собой вызов функции-члена шаблона класса std::basic_ostream, тогда как второй – это вызов функции-нечлена перегруженной функции operator<< пространства имен org_semantics. Такие подробности на самом деле совершенно неинтересны автору приветствия, а ADL справляется со всем этим просто замечательно.

Тема 26 | Поиск операторной функции

Иногда все выглядит так, как будто операторная функция-член перегружает оператор-нечлен. Но это не так. Это не перегрузка, а просто другой алгоритм поиска. Рассмотрим следующий класс, перегружающий оператор как функцию-член:

```
class X {
   public:
      X operator %( const X & ) const; // бинарный оператор нахождения модуля
      X memFunc1( const X & );
      void memFunc2();
      //...
};
```

Перегруженную операторную функцию можно вызвать или как инфиксный оператор, или с помощью синтаксиса вызова функции:

```
X a, b, c;
a = b % c;              // инфиксный вызов оператора-члена %
a = b.operator %( c );  // вызов функции-члена
a = b.memFunc1( c );    // вызов другой функции-члена
```

Синтаксис вызова функции активизирует обычные правила поиска (см. тему 24 «*Поиск функции-члена*») и вызов `b.operator %(c)` интерпретируется так же, как и аналогичный вызов `memFunc1`. Однако инфиксный вызов перегруженного оператора обрабатывается иначе:

```
X operator %( const X &, int ); // оператор-нечлен
//...
void X::memFunc2() {
   *this % 12;                  // вызывает оператор-нечлен %
   operator %( *this, 12 );     // ошибка! слишком много аргументов
}
```

Для вызова инфиксного оператора компилятор рассмотрит обе функции, и член и нечлен (см. также тему 25 «*Поиск, зависимый от типов аргументов*»). Таким образом, первый инфиксный вызов `operator %` будет соответствовать нечлену. Это не экземпляр перегрузки, просто компилятор проводит поиск функций в двух разных местах. Далее неинфиксный вызов подчиняется стандартным правилам поиска функций и выявляет функцию-член. Мы получили ошибку, потому что пытались передать три аргумента в бинарную функцию. (Вспомните неявный аргумент `this` для функций-членов!)

В сущности инфиксные вызовы перегруженных операторов реализуют своего рода «вырожденный» ADL, в котором при выборе функции, используемой для разрешения перегрузки, учитываются и класс левого (или единственного) аргумента инфиксного оператора, и глобальная область видимости. ADL расширяет этот процесс операторными функциями других пространств имен, внесенных аргументами оператора. Обратите внимание, что это не перегрузка. Перегрузка – это статическое свойство объявления функции (см. тему 21 «*Перегрузка и переопределение – это не одно и то же*»). И ADL, и поиск инфиксного оператора – это свойства аргументов вызова функции.

Тема 27 | Запросы возможностей

В большинстве случаев используемый объект способен выполнять все, что от него требуется, потому что его возможности явно проанонсированы в его интерфейсе. В таких случаях мы не спрашиваем объект, может ли он сделать ту или иную работу; мы просто приказываем ему приступить к ней:

```
class Shape {
   public:
      virtual ~Shape();
      virtual void draw() const = 0;
      //...
};
//...
Shape *s = getSomeShape();  // прими фигуру и скажи ей...
s->draw();                  // ...за работу!
```

Конечно, мы не знаем точно, с каким типом фигуры имеем дело, но знаем, что это Shape и, следовательно, может отрисовать себя. Так все и обстоит, просто и рационально, а нам того и надо.

Однако жизнь не всегда настолько проста. Иногда возможности объекта не столь очевидны. Пусть, например, необходима фигура, которая может катиться:

```
class Rollable {
   public:
      virtual ~Rollable();
      virtual void roll() = 0;
};
```

Такие классы, как Rollable (то, что можно катать), часто называют *интерфейсными классами*, потому что они определяют только интерфейс,

подобно Java-интерфейсу. Обычно у таких классов нет нестатических членов данных, нет объявленного конструктора, есть виртуальный деструктор и набор чисто виртуальных функций, которые определяют, что может делать объект Rollable. В данном случае говорится: все, что является Rollable, может катиться; все остальное не может:

```
class Circle : public Shape, public Rollable {  // круги могут катиться
   //...
   void draw() const;
   void roll();
   //...
};
class Square : public Shape {                  // квадраты не могут
   //...
   void draw() const;
   //...
};
```

Конечно, кроме фигур, могут катиться и другие объекты:

```
class Wheel : public Rollable { ... };
```

В идеале код должен быть организован таким образом, чтобы еще до попыток вращения (применения метода roll()) всегда было известно, относится ли объект к классу Rollable – как раньше мы знали, что имеем дело с объектом типа Shape до попытки отрисовать его.

```
vector<Rollable *> rollingStock;
//...
for( vector<Rollable *>::iterator i( rollingstock.begin() );
                                i != rollingStock.end(); ++i )
    (*i)->roll();
```

К сожалению, время от времени возникают ситуации, когда просто неизвестно, обладает ли объект требуемой возможностью. В таких случаях приходится осуществлять запрос возможностей. В C++ запрос возможностей обычно представляется в виде dynamic_cast (динамическое приведение) между несвязанными типами (см. тему 9 «*Новые операторы приведения*»).

```
Shape *s = getSomeShape();
Rollable *roller = dynamic_cast<Rollable *>(s);
```

Этот вид dynamic_cast часто называют *перекрестным приведением* (*cross-cast*), потому что здесь происходит превращение на одном уровне иерархии, а не вверх или вниз по ней (рис. 27.1).

Если s ссылается на Square, dynamic_cast даст сбой (будет получен нулевой указатель), а значит, тип Shape, на который ссылается s, не является Rol-

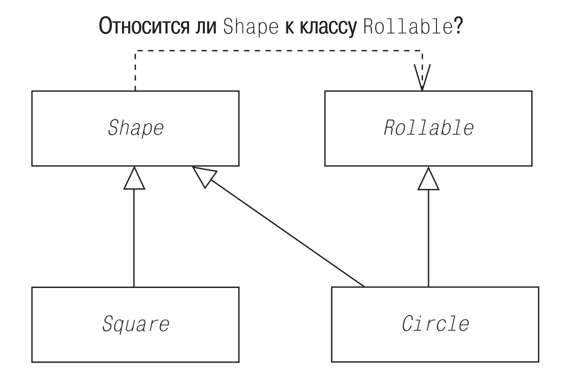


Рис. 27.1. *Запрос возможности: «Эту фигуру можно катить?»*

lable. Если s ссылается на Circle или любой другой тип Shape, производный от Rollable, приведение пройдет успешно, и мы будем знать, что можем вращать эту фигуру.

```
Shape *s = getSomeShape();
if( Rollable *roller = dynamic_cast<Rollable *>(s) )
    roller->roll();
```

Необходимость в запросах возможностей время от времени возникает, но часто ими злоупотребляют. Обычно они являются индикатором плохого проектирования. Если доступны другие рациональные подходы, лучше избегать применения запросов времени выполнения о возможностях объектов.

Тема 28 | Смысл сравнения указателей

В C++ у объекта может быть несколько действительных адресов. Сравнение указателей на объекты – это не вопрос об адресах. Это вопрос об идентичности объектов.

```
class Shape { ... };
class Subject { ... };
class ObservedBlob : public Shape, public Subject { ... };
```

В приведенной иерархии ObservedBlob является производным и от Shape, и от Subject, и (поскольку наследование открытое) существуют предопределенные преобразования из ObservedBlob в любой из его базовых классов.

```
ObservedBlob *ob = new ObservedBlob;
Shape *s = ob;          // предопределенное преобразование
Subject *subj = ob;     // предопределенное преобразование
```

Доступность этих преобразований означает, что указатель на ObservedBlob можно сравнивать с указателем на любой из его базовых классов.

```
if( ob == s ) ...
if( subj == ob ) ...
```

В данном случае оба эти условия будут истинными, даже если адреса, содержащиеся в ob, s и subj, разные. Рассмотрим два возможных размещения в памяти объекта ObservedBlob, на который ссылаются эти указатели (рис. 28.1).

При первом размещении s и subj ссылаются на подобъекты Shape и Subject полного объекта, адреса которых отличаются от адреса полного объекта, на который ссылается ob. При втором размещении подобъект Shape имеет тот же адрес, что и полный объект ObservedBlob, таким образом, ob и s содержат один и тот же адрес.

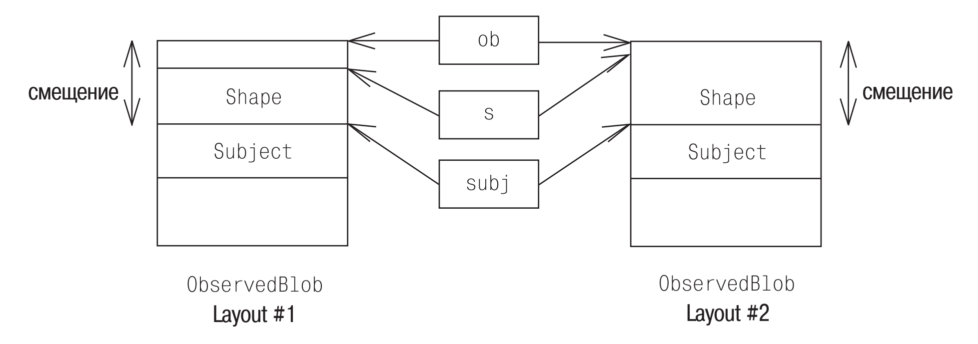


Рис. 28.1. *Два возможных размещения объекта при множественном наследовании. При любом размещении объект имеет несколько адресов*

При любом размещении `ob`, `s` и `subj` ссылаются на один и тот же объект `ObservedBlob`, поэтому компилятор должен убедиться, что `ob` идентичен и `s`, и `subj`. (Нельзя сравнивать `s` с `subj`, потому что между ними нет отношения наследования.) Это сравнение компилятор осуществляет путем дополнения значения одного из сравниваемых указателей соответствующим смещением. Например, выражение

```
ob == subj
```

может быть интерпретировано (вольно) так:

```
ob ? (ob+delta == subj) : (subj == 0)
```

где `delta` — смещение подобъекта `Subject` в `ObservedBlob`. Другими словами, `ob` и `subj` равны, если обе они нулевые; в противном случае `ob` настраивается, чтобы ссылаться на подобъект базового класса `Subject`, и затем сравнивается с `subj`.

Из этих рассуждений необходимо извлечь один важный урок: при работе с указателями и ссылками на объекты (да и вообще всегда) необходимо быть осторожными, чтобы не потерять информацию о типе. Указатели на `void` — обычные виновники этой потери:

```
void *v = subj;
if( ob == v )     // не равны!
```

Как только содержащийся в `subj` адрес оказался лишенным информации о своем типе из-за копирования в `void *`, компилятору не остается ничего другого, как только прибегнуть к простому сравнению адресов. Для указателей на объекты классов это приемлемо далеко не всегда.

Тема 29 | Виртуальные конструкторы и Прототип

Представьте, что вы сидите в шведском ресторане и хотите сделать заказ. К несчастью, ваше знание шведского ограничено технической документацией, ругательствами или (как правило) комбинацией одного и другого. Меню написано на шведском, читать на котором вы не умеете. Однако в другом конце комнаты вы замечаете джентльмена, наслаждающегося своим кушаньем. Поэтому вы подзываете официанта и говорите:

> *«Если тот джентльмен ест рыбу, я бы хотел рыбу. Если он ест спагетти, я тоже хочу спагетти. Если же он ест угря, пусть будет угорь. Даже если он остановился на консервированных кумкватах, то и мне дайте того же».*

Разве это звучит разумно? Конечно, нет. (Так, например, вряд ли следует заказывать спагетти в шведском ресторане.) В данной ситуации можно обозначить две основные проблемы. Во-первых, этот метод утомителен и неэффективен. Во-вторых, он может не сработать. А если, закончив свою речь, вы так и не угадаете, что ест другой посетитель? Официант уйдет, оставив вас в затруднительном положении и голодным. Даже если вам каким-то чудом удалось узнать все меню и поэтому успех (в конечном счете) гарантирован, список вопросов может оказаться недействительным или неполным – ведь меню могли изменить в промежутке между вашими посещениями ресторана.

Правильно было бы, конечно, просто подозвать официанта и сказать: «Я бы хотел то, что ест тот джентльмен».

Если официант привык все понимать буквально, он выхватит наполовину съеденное блюдо у другого посетителя и принесет его вам. Однако такое поведение может обидеть и даже привести к конфликту. Подобные

недоразумения происходят, когда два посетителя пытаются взять одно и то же блюдо одновременно. Если официант знает свое дело, он принесет вам точную копию блюда, поглощаемого другим клиентом, никак не влияя на состояние копируемого блюда.

Мы проиллюстрировали два основных мотива клонирования, которые можно сформулировать так: вы не обязаны знать точный тип объекта, с которым имеете дело, и не хотите изменять исходный объект или зависеть от его изменений.

Функцию-член, предоставляющую возможность клонирования объекта, в C++ традиционно называют *виртуальным конструктором*. Конечно, виртуальных конструкторов не существует, но при создании копии объекта обычно осуществляется непрямой вызов конструктора копий класса через виртуальную функцию, что создает эффект – если не реальность – виртуального конструктора. В последнее время эту технику называют *экземпляром паттерна Прототип* (*Prototype*).

Конечно, надо *хоть что-то* знать об объекте, на который осуществляется ссылка. В данном случае известно, что нам нужна еда (meal).

```
class Meal {
   public:
      virtual ~Meal();
      virtual void eat() = 0;
      virtual Meal *clone() const = 0;
      //...
};
```

Тип `Meal` обеспечивает возможность клонирования с помощью функции-члена `clone`. Функция `clone` представляет собой специализированную разновидность Фабричного метода (см. тему 30 «*Фабричный метод*»), которая создает соответствующий продукт, при этом обеспечивая возможность инициирующему коду оставаться в неведении о точном типе контекста и классе продукта. Конкретные классы, производные от `Meal` (те блюда, которые фактически существуют и перечислены в меню), должны предоставлять реализацию чисто виртуальной операции клонирования.

```
class Spaghetti : public Meal {
   public:
      Spaghetti( const Spaghetti & );      // конструктор копий
      void eat();
      Spaghetti *clone() const
         { return new Spaghetti( *this ); } // вызов конструктора копий
   //...
};
```

(Почему возвращаемый тип переопределяющей функции clone производного класса отличается от возвращаемого типа функции базового класса, объясняется в теме 31 «*Ковариантные возвращаемые типы*».)

С помощью этого простого каркаса можно создавать копии любого типа Meal, не зная точно фактического типа копируемого Meal. Обратите внимание, что в следующем коде нет никакого упоминания о конкретных производных классах и, следовательно, нет привязки кода к какому-либо текущему *или будущему* производному от Meal типу.

```
const Meal *m = thatGuysMeal();    // что бы это ни было...
Meal *myMeal = m->clone();         // ...я тоже хочу это!
```

Фактически можно дойти до того, чтобы заказывать то, о чем никогда даже не слышал. Благодаря использованию паттерна Прототип незнание о существовании типа не мешает создать объект этого типа. Приведенный выше полиморфный код может быть откомпилирован, распределен и в дальнейшем пополнен новыми типами Meal без необходимости повторной компиляции.

Этот пример иллюстрирует некоторые преимущества неведения при проектировании программного обеспечения, в частности, структурированного как каркас, предназначенный для настройки и расширения. Чем меньше знаешь, тем лучше.

Тема 30 | Фабричный метод

Высокоуровневое проектирование часто требует создания объекта «соответствующего» типа на основании типа существующего объекта. Например, имеется указатель или ссылка на объект `Employee` (Служащий) какого-то типа. Необходимо сгенерировать соответствующий для этого типа `Employee` объект `HRInfo` (Сведения о ресурсе), как показано на рис. 30.1.

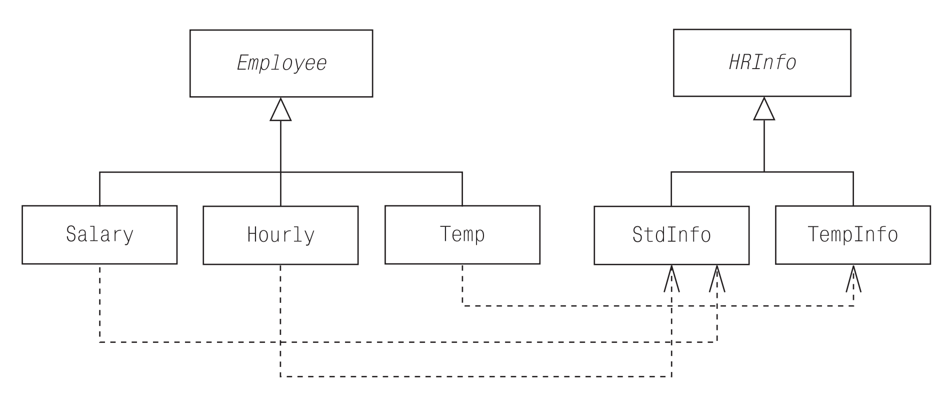


Рис. 30.1*. Псевдопараллельные иерархии. Как проецируется служащий на соответствующую информацию о трудовых ресурсах?*

Здесь иерархии `Employee` и `HRInfo` практически параллельны. Служащим типа `Salary` (Оклад) и `Hourly` (Почасовая оплата) требуются объекты типа `StdInfo`, тогда как служащим типа `Temp` (Временный) требуется объект `TempInfo`.

Идея высокоуровневого проектирования проста: необходимо создать соответствующий тип записи для этого служащего. К сожалению, программисты нередко считают, что это оправдывает использование запросов

о типе во время выполнения. То есть код, реализующий это требование, определяет тип генерируемого объекта HRInfo, задавая вопросы о точном типе Employee.

Использование кода типа служащего и инструкции switch — распространенный и всегда ошибочный подход:

```
class Employee {
  public:
    enum Type { SALARY, HOURLY, TEMP };
    Type type() const { return type_; }
    //...
  private:
    Type type_;
    //...
};
//...
HRInfo *genInfo( const Employee &e ) {
    switch( e.type() ) {
    case SALARY:
    case HOURLY: return new StdInfo( e );
    case TEMP:return new TempInfo( static_cast<const Temp *>(e) );
    default: return 0;    // код неизвестного типа!
    }
}
```

Ничем не лучше использование dynamic_cast для задания серии личных вопросов об объекте Employee:

```
HRInfo *genInfo( const Employee &e ) {
    if( const Salary *s = dynamic_cast<const Salary *>(&e) )
        return new StdInfo( s );
    else if( const Hourly *h = dynamic_cast<const Hourly *>(&e) )
        return new StdInfo( h );
    else if( const Temp *t = dynamic_cast<const Temp *>(&e) )
        return new TempInfo( t );
    else
        return 0;           // неизвестный тип служащего!
}
```

Основная ошибка в обеих приведенных реализациях genInfo в том, что они связаны с конкретными производными от Employee и HRInfo типами и должны уметь проецировать служащих всех типов в соответствующие типы HRInfo. Любые изменения в множестве типов Employee, HRInfo или в проецировании одного в другое требуют изменения кода. Скорее всего новые типы в этих иерархиях будут добавляться (или удаляться) постоянно; маловероятно, что изменения в коде всегда будут осуществляться

правильно. Другая проблема – при любом подходе может произойти ошибка в определении точного типа аргумента Employee. При этом потребуется организовать обработку этой ошибки в коде, вызывающем genInfo().

Верный подход состоит в том, чтобы рассмотреть, где должна находиться проекция из любого типа Employee на соответствующий тип HRInfo. Иначе говоря, кто лучше всего знает, какой тип объекта HRInfo нужен служащему типа Temp? Конечно, сам служащий Temp:

```
class Temp : public Employee {
   public:
      //...
      TempInfo *genInfo() const
         { return new TempInfo( *this ); }
      //...
};
```

По-прежнему остается нерешенной проблема, связанная с тем, что может быть неизвестен тип служащих, с которыми мы имеем дело. Но это легко исправить с помощью виртуальной функции:

```
class Employee {
   public:
      //...
      virtual HRInfo *genInfo() const = 0; // Фабричный метод
      //...
};
```

Это экземпляр паттерна Фабричный метод (Factory Method). Вместо массы глупых личных вопросов, по сути, служащему говорят: «Какого бы типа ты ни был, генерируй для себя соответствующий тип информации».

```
Employee *e = getAnEmployee();
//...
HRInfo *info = e->genInfo();         // используется Фабричный метод
```

Суть Фабричного метода в том, что базовый класс предоставляет виртуальную функцию-ловушку для генерирования соответствующего «продукта». Каждый производный класс может переопределить эту унаследованную виртуальную функцию, чтобы генерировать собственный продукт. В результате получаем возможность использования объекта неизвестного типа («некоторый тип служащих») для генерирования объекта неизвестного типа («соответствующий тип информации»).

Обычно о необходимости применения Фабричного метода при высокоуровневом проектировании свидетельствует потребность генерировать «соответствующий» объект на основании конкретного типа другого объекта, параллельные или почти параллельные иерархии. Часто этот паттерн является лекарством от серии запросов о типе на этапе выполнения.

Тема 31 | Ковариантные возвращаемые типы

Как правило, возвращаемый тип переопределяющей функции должен быть таким же, как и у функции, которую она переопределяет:

```
class Shape {
   public:
      //...
      virtual double area() const = 0;
      //...
};
class Circle : public Shape {
   public:
      float area() const;   // ошибка! другой возвращаемый тип
      //...
};
```

Однако для так называемых *ковариантных возвращаемых типов* это правило не настолько строгое. Если `B` — тип класса, и виртуальная функция базового класса возвращает `B *`, переопределяющая функция производного класса может возвращать `D *`, где `D` — открыто наследуется от `B`. (То есть `D` представляет собой `B`.) Если виртуальная функция базового класса возвращает `B &`, переопределяющая функция производного класса может возвращать `D &`. Рассмотрим следующую операцию клонирования в иерархии фигур (класса `Shape`) (см. тему 29 «*Виртуальные конструкторы и Прототип*»):

```
class Shape {
   public:
      //...
      virtual Shape *clone() const = 0;  // Прототип
      //...
};
```

```
class Circle : public Shape {
   public:
      Circle *clone() const;
      //...
};
```

Объявлено, что переопределяющая функция производного класса возвращает `Circle *`, а не `Shape *`. Это допустимо, потому что `Circle` представляет собой `Shape`. Обратите внимание, что возвращаемое из `Circle::clone` значение `Circle *` автоматически преобразуется в `Shape *`, если `Circle` интерпретируется как `Shape` (см. тему 28 «*Смысл сравнения указателей*»):

```
Shape *s1 = getACircleOrOtherShape();
Shape *s2 = s1->clone();
```

Преимущество применения ковариантных возвращаемых типов становится очевидным при работе с производными типами напрямую, а не через интерфейсы их базового класса:

```
Circle *c1 = getACircle();
Circle *c2 = c1->clone();
```

Без ковариантного возвращаемого типа `Circle::clone` должен был бы точно совпадать с возвращаемым типом `Shape::clone` и возвращать `Shape *`. Пришлось бы приводить возвращаемый результат к `Circle *`.

```
Circle *c1 = getACircle();
Circle *c2 = static_cast<Circle *>(c1->clone());
```

В качестве другого примера рассмотрим следующий член `Shape`, относящийся к Фабричному методу, который возвращает ссылку на соответствующий редактор конкретной фигуры (см. тему 30 «*Фабричный метод*»):

```
class ShapeEditor { ... };
class Shape {
   public:
      //...
      virtual const ShapeEditor &
         getEditor() const = 0; // Фабричный метод
      //...
};
//...
class Circle;
class CircleEditor : public ShapeEditor { ... };
class Circle : public Shape {
   public:
      const CircleEditor &getEditor() const;
      //...
};
```

В данном случае обратите внимание, что `CircleEditor` должен быть полностью определен (не просто объявлен) до объявления `Circle::getEditor`. Компилятор должен знать устройство объекта `CircleEditor`, чтобы иметь возможность осуществлять соответствующие манипуляции с адресом для преобразования ссылки (или указателя) на `CircleEditor` в ссылку (или указатель) на `ShapeEditor`. (См. тему 28 «*Смысл сравнения указателей*».)

Преимущество ковариантного возвращаемого типа в том, что всегда можно работать на соответствующем уровне абстракции. Если работаешь с объектами `Shape`, получаешь абстрактный `ShapeEditor`; работая с определенным типом фигуры, например `Circle`, можешь иметь дело напрямую с `CircleEditor`. Ковариантный возвращаемый тип освобождает от необходимости применять приведение (так чреватое ошибками) для допоставки информации о типе, которая ни в коем случае не должна быть утеряна:

```
Shape *s = getACircleOrOtherShape();
const ShapeEditor &sed = s->getEditor();
Circle *c = getACircle();
const CircleEditor &ced = c->getEditor();
```

Тема 32 | Предотвращение копирования

Спецификаторы доступа (public, protected и private) могут применяться для выражения и реализации высокоуровневых ограничений на использование типа. Чаще всего для этого запрещают копировать объект. Операции копирования объекта объявляются закрытыми и не определяются:

```
class NoCopy {
   public:
      NoCopy( int );
      //...
   private:
      NoCopy( const NoCopy & );              // конструктор копий
      NoCopy &operator =( const NoCopy & );  // копирующее присваивание
};
```

Конструктор копий и оператор копирующего присваивания должны быть обязательно объявлены, поскольку в противном случае компилятор объявит их неявно как открытые подставляемые в строку члены. Объявление их закрытыми предотвращает вмешательство компилятора и гарантирует, что любое использование этих операций – явное или неявное – приведет к ошибке компиляции:

```
void aFunc( NoCopy );
void anotherFunc( const NoCopy & );
NoCopy a( 12 );
NoCopy b( a ); // ошибка! конструктор копий
NoCopy c = 12; // ошибка! неявный конструктор копий
a = b;         // ошибка! копирующее присваивание
aFunc( a );  // ошибка! передача по значению с помощью конструктора копий
aFunc( 12 );   // ошибка! неявный конструктор копий
anotherFunc( a ); // OK, передача по ссылке
anotherFunc( 12 ); // OK
```

Тема 33 | Как сделать базовый класс абстрактным

Абстрактные базовые классы обычно представляют абстрактные концепции предметной области, поэтому нет смысла объявлять объекты этих типов. Абстрактным базовый класс можно сделать, объявив (или унаследовав) по крайней мере одну чисто виртуальную функцию. Тогда компилятор гарантирует невозможность создания объекта абстрактного базового класса.

```
class ABC {
   public:
      virtual ~ABC();
      virtual void anOperation() = 0;  // чисто виртуальная
      //...
};
```

Однако иногда подходящего кандидата на чисто виртуальную функцию нет, но необходимо, чтобы класс вел себя как абстрактный базовый. В этих случаях приблизиться к природе абстрактности можно, убрав из класса все открытые конструкторы. Это неизменно означает необходимость явного объявления по крайней мере одного конструктора, поскольку в противном случае компилятор неявно объявит открытый подставляемый конструктор по умолчанию. Компилятор также объявит неявный конструктор копий (если он не объявлен явно), поэтому обычно приходится объявлять два конструктора.

```
class ABC {
   public:
      virtual ~ABC();
   protected:
```

```
        ABC();
        ABC( const ABC & );
        //...
};
class D : public ABC {
        //...
};
```

Конструкторы объявляются защищенными, чем обеспечивается возможность их использования конструкторами производных классов, но предотвращается создание отдельных объектов ABC.

```
void func1( ABC );
void func2( ABC & );
ABC a;          // ошибка! защищенный конструктор по умолчанию
D d;            // OK
func1( d );     // ошибка! защищенный конструктор копий
func2( d );     // OK, нет конструктора копий
```

Другой способ сделать класс абстрактным базовым – назначить одну из его виртуальных функций чисто виртуальной. Обычно для этого отлично подходит деструктор:

```
class ABC {
   public:
      virtual ~ABC() = 0;
      //...
};
//...
ABC::~ABC() { ... }
```

Заметьте, что в данном случае необходимо предоставить реализацию чисто виртуальной функции, поскольку деструкторы производных классов будут вызывать деструкторы своего базового класса неявно. (Обратите внимание, что этот неявный вызов деструктора базового класса из деструктора производного класса всегда является невиртуальным.)

Третий подход применяется, когда у класса вообще нет виртуальных функций и нет необходимости явного объявления конструкторов. В этом случае целесообразно создать защищенный невиртуальный деструктор.

```
class ABC {
   protected:
      ~ABC();
   public:
      //...
};
```

Эффект от применения защищенного деструктора практически такой же, как и от защищенного конструктора, но ошибка возникает не при создании объекта, а когда объект выходит за рамки области видимости или явно уничтожается:

```
void someFunc() {
   ABC a;            // ошибки пока нет...
   ABC *p = new ABC; // ошибки пока нет...
   //...
   delete p; // ошибка! защищенный деструктор
   // ошибка! неявный вызов деструктора a
}
```

Тема 34 | Ограничение на размещение в куче

Иногда требуется показать, что объекты определенного класса не должны размещаться в куче (heap). Часто это делается для того, чтобы гарантировать вызов деструктора объекта, как в случае с объектом «handle» (дескриптор), реализующим подсчет ссылок для объекта «body» (тело). Деструкторы локальных объектов классов с автоматическим распределением памяти будут вызываться автоматически, кроме случаев аварийного прерывания программы с помощью функций `exit` (выход) или `abort` (преждевременное прекращение). То же самое происходит и с объектами статически создаваемых классов (кроме выхода по функции `abort`), тогда как динамически создаваемые объекты должны уничтожаться явно.

Обозначить такое предпочтение можно через запрещение выделения памяти в куче:

```
class NoHeap {
   public:
      //...
   protected:
      void *operator new( size_t ) { return 0; }
      void operator delete( void * ) {}
};
```

Любая стандартная попытка размещения объекта `NoHeap` в куче приведет к ошибке компиляции (см. тему 36 «*Индивидуальное управление памятью*»):

```
NoHeap *nh = new NoHeap;  // ошибка! защищенный NoHeap::operator new
//...
delete nh;                // ошибка! защищенный NoHeap::operator delete
```

Члены operator new и operator delete определены (и объявлены), потому что на некоторых платформах конструкторы и деструкторы могут вызывать их неявно. По той же причине они объявляются защищенными: они могут быть инициированы неявно конструкторами и деструкторами производных классов. Если NoHeap не предполагается использовать как базовый класс, эти функции можно объявить закрытыми.

В то же время может понадобиться уделить внимание размещению массивов объектов NoHeap (см. тему 37 «*Создание массивов*»). В этом случае достаточно объявить функции создания и уничтожения массива закрытыми и неопределенными, аналогично тому, как налагается запрет на операции копирования (см. тему 32 «*Предотвращение копирования*»).

```
class NoHeap {
   public:
      //...
   protected:
      void *operator new( size_t ) { return 0; }
      void operator delete( void * ) {}
   private:
      void *operator new[]( size_t );
      void operator delete[]( void * );
};
```

Кроме того, можно содействовать размещению в куче, а не запрещать его. Просто делаем деструктор закрытым:

```
class OnHeap {
      ~OnHeap();
   public:
      void destroy()
         { delete this; }
      //...
};
```

Любое обычное объявление объекта с автоматическим распределением памяти или статически создаваемого объекта OnHeap приводит к неявному вызову деструктора, когда имя объекта выходит за рамки области видимости.

```
OnHeap oh1; // ошибка! неявный вызов закрытого деструктора
void aFunc() {
   OnHeap oh2;
   //...
   // ошибка! неявный вызов деструктора для oh2
}
```

Тема 35 | Синтаксис размещения new

Вызвать конструктор напрямую невозможно. Однако с помощью синтаксиса размещения new можно обмануть компилятор и заставить его вызывать конструктор.

```
void *operator new( size_t, void *p ) throw()
      { return p; }
```

Синтаксис размещения new – это стандартная глобальная перегруженная версия operator new, которая не может быть замещена пользовательской версией (в отличие от стандартных глобальных «обычных» operator new и operator delete, которые могут замещаться, но это не рекомендуется). Реализация игнорирует аргумент размера и возвращает второй аргумент. Тем самым разрешается «размещать» объект в определенном месте, что обеспечивает эффект возможности вызова конструктора.

```
class SPort { ... };            // представляет последовательный порт
const int comLoc = 0x00400000;  // местоположение порта
//...
void *comAddr = reinterpret_cast<void *>(comLoc);
SPort *com1 = new (comAddr) SPort; // создается объект по адресу comLoc
```

Важно отличать оператор new от функций под именем operator new. Оператор new не может быть перегружен и, таким образом, всегда ведет себя одинаково: вызывает функцию operator new и инициализирует возвращаемую область памяти. Все варианты поведения при распределении памяти обеспечиваются разными перегруженными версиями operator new, а не оператором new. То же самое можно сказать и об операторе delete и функции operator delete.

Синтаксис размещения new – разновидность функции operator new, которая на самом деле не выделяет память. Она лишь возвращает указатель

на хранилище, которое (предположительно) уже существует. Поскольку при вызове синтаксиса размещения new память не выделяется, важно не удалить его.

```
delete com1;    // ой!
```

Однако, хотя память и не была выделена, объект был создан, и этот объект должен быть уничтожен по окончании его времени жизни. Мы избегаем оператора delete и вместо него вызываем прямо деструктор объекта:

```
com1->~SPort();  // вызывается деструктор, а не оператор delete
```

Однако часто прямые вызовы деструкторов сопровождаются ошибками, в результате чего либо один и тот же объект уничтожается многократно, либо объект не уничтожается вовсе. Такие конструкции лучше применять только в случае необходимости в хорошо скрытых и правильно используемых областях кода.

Также существует синтаксис размещения массива, позволяющий создавать массив объектов в заданном местоположении:

```
const int numComs = 4;
//...
SPort *comPorts = new (comAddr) SPort[numComs]; // создается массив
```

Конечно, эти элементы массива в конце концов должны быть уничтожены:

```
int i = numComs;
while( i )
   comPorts[--i].~SPort();
```

Для массивов объектов класса характерна одна проблема – при размещении массива каждый элемент должен быть инициализирован вызовом конструктора по умолчанию. Рассмотрим простой буфер фиксированного размера, в который может быть добавлено новое значение:

```
string *sbuf = new string[BUFSIZE];  // вызывается конструктор по умолчанию
                                     // для всех элементов массива!
int size = 0;
void append( string buf[], int &size, const string &val )
    { buf[size++] = val; }      // уничтожает инициализацию по умолчанию!
```

Если используется только часть массива или если элементам немедленно присваиваются значения, это решение неэффективно. Еще хуже, если тип элементов массива не имеет конструктора по умолчанию. Тогда мы получим ошибку компиляции.

Синтаксис размещения new часто применяется для решения этой проблемы с буфером. При таком подходе хранилище для буфера выделяется та-

ким образом, чтобы избежать инициализации конструктором по умолчанию (если таковой существует).

```
const size_t n = sizeof(string) * BUFSIZE;
string *sbuf = static_cast<string *>(::operator new( n ));
int size = 0;
```

Нельзя осуществить присваивание при первом обращении к элементу массива, потому что он еще не был инициализирован (см. тему 12 «*Присваивание и инициализация – это не одно и то же*»). Поэтому для инициализации элемента конструктором копий применяется синтаксис размещения `new`:

```
void append( string buf[], int &size, const string &val )
    { new (&buf[size++]) string( val ); } // синтаксис размещения new
```

Как обычно, при использовании синтаксиса размещения `new` требуется провести очистку ресурсов самостоятельно:

```
void cleanupBuf( string buf[], int size ) {
    while( size )
        buf[--size].~string();  // уничтожаем инициализированные элементы
    ::operator delete( buf );   // высвобождаем хранилище
}
```

Данный подход быстр, разумен и не предназначен для широкой публики. Описанная базовая методика широко применяется (в более сложной форме) в большинстве реализаций контейнеров стандартной библиотеки.

Тема 36 | Индивидуальное управление памятью

Стандартные функции operator new и operator delete не всегда обрабатывают типы классов так, как нам надо. Но у типов могут быть собственные функции operator new и operator delete, соответствующие их нуждам.

Мы ничего не можем сделать с операторами new или delete, потому что их поведение неизменно, но можно изменить вызываемые ими функции operator new и operator delete (см. тему 35 «*Синтаксис размещения new*»). Лучше всего объявить функции-члены operator new и operator delete:

```
class Handle {
   public:
      //...
      void *operator new( size_t );
      void operator delete( void * );
      //...
};
//...
Handle *h = new Handle; // используем Handle::operator new
//...
delete h;               // используем Handle::operator delete
```

При размещении в памяти объекта типа Handle в выражении new компилятор сначала поищет operator new в области видимости Handle. Если член operator new не будет найден, компилятор использует глобальный operator new. Аналогична ситуация и для operator delete, поэтому обычно имеет смысл определять член operator delete, если определяется operator new, и наоборот.

Члены operator new и operator delete представляют собой статические функции-члены (см. тему 63 «*Необязательные ключевые слова*»), что

оправданно. Вспомним, что у статических функций-членов нет указателя this. Они отвечают только за получение и высвобождение хранилища для объекта, поэтому им не нужен указатель this. Как и другие функции-члены, они наследуются производными классами:

```
class MyHandle : public Handle {
   //...
};
//...
MyHandle *mh = new MyHandle;  // используется Handle::operator new
//...
delete mh;                    // используется Handle::operator delete
```

Конечно, если бы MyHandle объявил собственные operator new и operator delete, компилятор находил бы их первыми при поиске, и они бы использовались вместо унаследованных от базового класса Handle версий.

При определении членов operator new и operator delete в базовом классе деструктор класса должен быть гарантированно виртуальным:

```
class Handle {
   public:
      //...
      virtual ~Handle();
      void *operator new( size_t );
      void operator delete( void * );
      //...
};
class MyHandle : public Handle {
      //...
      void *operator new( size_t );
      void operator delete( void *, size_t ); // обратите внимание
                                              // на 2-й аргумент
      //...
};
//...
Handle *h = new MyHandle; // используется MyHandle::operator new
//...
delete h;                 // используется MyHandle::operator delete
```

Без виртуального деструктора эффект от удаления объекта производного класса через указатель на базовый класс непредсказуем! Реализация может (и, вероятно, неправильно) инициировать Handle::operator delete, а не MyHandle::operator delete. Вообще случиться может все, что угодно. Также обратите внимание, что здесь реализован двухаргументный вариант operator delete, а не обычный одноаргументный. Двухаргументная версия представляет собой еще одну «обычную» версию члена operator delete. Ее часто используют базовые классы, для которых предполагается наследование

производными классами их реализации operator delete. Второй аргумент будет содержать размер удаляемого объекта. Эта информация часто оказывается полезной при реализации обычного распределения памяти.

Широко распространено заблуждение, что применение операторов new и delete подразумевает необходимость работать с кучей (heap), или свободной памятью. Вызов оператора new лишь означает, что будет вызвана функция operator new и эта функция возвратит указатель на некоторую область памяти. Стандартные глобальные operator new и operator delete действительно распределяют память из кучи, но члены operator new и operator delete могут делать все, что угодно. Нет ограничения на то, где будет выделена область памяти. Она может поступить из специальной кучи, из статически выделенного блока, из недр стандартного контейнера или из блока локальной памяти функции. Единственным ограничением в данном случае являются изобретательность разработчика и здравый смысл. Например, объекты Handle могли бы быть размещены в таком статическом блоке:

```
struct rep {
   enum { max = 1000 };
   static rep *free;       // начало статического блока
   static int num_used;    // число используемых слотов
   union {
      char store[sizeof(Handle)];
      rep *next;
   };
};
static rep mem[ rep::max ];            // блок статического хранилища
void *Handle::operator new( size_t ) {
   if( rep::free ) {                   // если есть слоты в списке free
      rep *tmp = rep::free;            // берем объект из списка free
      rep::free = rep::free->next;
      return tmp;
   }
   else if( rep::num_used < rep::max ) // если остались слоты
      return &mem[ rep::num_used++ ];  // вернуть неиспользованный слот
   else                                // в противном случае,...
      throw std::bad_alloc();          // ...нехватка памяти!
}
void Handle::operator delete( void *p ) { // добавить в список free
   static_cast<rep *>(p)->next = rep::free;
   rep::free = static_cast<rep *>(p);
}
```

Промышленная версия была бы более строгой относительно условий нехватки памяти при работе с производными от Handle типами, массивами объектов Handle и т. д. Тем не менее этот код демонстрирует, что операторам new и delete необязательно работать с кучей.

Тема 37 | Создание массивов

Большинство программистов на C++ знают, что необходимо строго придерживаться установленного синтаксиса для массивов и немассивов при выделении и высвобождении памяти.

```
T *aT = new T;       // немассив
T *aryT = new T[12]; // массив
delete [] aryT;      // массив
delete aT;           // немассив
aT = new T[1];       // массив
delete aT;           // ошибка! должен быть массив
```

Очень важно применять эти пары функций правильно, потому что при выделении и высвобождении памяти под массивы применяются совершенно другие функции, чем для немассивов. Для выделения памяти под создаваемый массив применяется не operator new, а его версия для массива. Аналогично для высвобождения памяти, занятой массивом, вызывается не operator delete, а его версия для массива. Точнее, при выделении памяти под массив используется один оператор (new[]), а при распределении памяти под немассив – другой (new). То же самое происходит при высвобождении памяти.

Функции создания и уничтожения массивов – это аналоги функций operator new и operator delete. Объявляются они аналогично:

```
void *operator new( size_t ) throw( bad_alloc );   // operator new
void *operator new[]( size_t ) throw( bad_alloc ); // версия new для массива
void operator delete( void * ) throw();            // operator delete
void operator delete[]( void * ) throw();           // версия delete для массива
```

Чаще всего путаница с формами этих функций для массивов возникает, когда отдельный класс или иерархия определяет собственное распределе-

ние памяти с помощью членов operator new и operator delete (см. тему 36 «*Индивидуальное управление памятью*»).

```
class Handle {
   public:
      //...
      void *operator new( size_t );
      void operator delete( void * );
      //...
};
```

Класс Handle определил функции распределения памяти для немассивов. Для массива объектов Handle будут вызваны не они, а глобальные функции создания и уничтожения массива:

```
Handle *handleSet = new Handle[MAX]; // вызывается ::operator new[]
//...
delete [] handleSet;                 // вызывается ::operator delete[]
```

Может показаться, что логично всегда объявлять эти функции в форме для массивов, даже для немассивов (хотя странно, что это не обычная практика). Если цель действительно состоит в вызове глобальных операций создания массива, было бы правильней, если бы локальные формы операторов просто делегировали такой вызов:

```
class Handle {
   public:
      //...
      void *operator new( size_t );
      void operator delete( void * );
      void *operator new[]( size_t n )
         { return ::operator new( n ); }
      void operator delete[]( void *p )
         { ::operator delete( p ); }
      //...
};
```

Если же надо препятствовать созданию массивов объектов Handle, тогда формы для массивов можно объявить закрытыми и оставить без определения (см. тему 34 «*Ограничение на размещение в куче*»).

Второй источник неразберихи и ошибок связан со значением аргумента размера, который передается в функцию создания массива в зависимости от того, как вызывается функция. Когда operator new вызывается (неявно) в выражении создания, компилятор определяет необходимый объем памяти и передает это значение как первый аргумент operator new. Это размер создаваемого объекта:

```
aT = new T; // вызывается operator new( sizeof(T) );
```

Также функция operator new может быть вызвана напрямую. В этом случае требуемое количество байтов надо задать явно:

```
aT = static_cast<T *>(operator new( sizeof(T) ));
```

Можно также напрямую вызвать версию operator new для массива:

```
aryT = static_cast<T *>(operator new[]( 5 * sizeof(T) ));
```

Однако при неявном вызове версии operator new для массива через выражение создания компилятор может немного увеличить запрашиваемое число байтов, и часто он именно так и поступает:

```
aryT = new T[5]; // запрашивается 5 * sizeof(T) + смещение
```

Дополнительный объем памяти обычно используется диспетчером рабочей памяти для записи информации о массиве, которая позже необходима для восстановления памяти (число размещенных элементов, размер каждого элемента и т. д.). Сложности возникают из-за того, что компилятор может запрашивать это дополнительное пространство не при каждом распределении памяти, и размер запроса может меняться от случая к случаю.

Настраивать запрашиваемый объем памяти обычно можно только в очень низкоуровневом коде, который размещает массивы напрямую. Если вы собираетесь заниматься разработкой кода нижнего уровня, то проще избегать прямых вызовов версии operator new для массива и связанных с этим действий компилятора и использовать старую добрую функцию operator new (см. тему 35 «*Синтаксис размещения new*»).

Тема 38 | Аксиомы надежности исключений

Написание надежной[1] программы немного напоминает доказательство теоремы евклидовой геометрии. Сначала с помощью минимального набора аксиом доказываются простые теоремы. Затем эти вспомогательные теоремы служат для доказательства более сложных полезных теорем. С надежностью то же самое. Надежный код создается из надежных компонентов. (Хотя любопытно отметить, что лишь композиция ряда надежных компонентов или вызовов функций не гарантирует, что результат будет надежным. Это было бы слишком просто, не правда ли?) Как в любой системе доказательств, здесь необходимо полагаться на ряд аксиом, с помощью которых создается наша надежная структура. Что это за аксиомы?

Аксиома 1: исключения синхронны

Во-первых, необходимо обратить внимание на то, что исключения синхронны и могут возникать только на границе вызова функции. Поэтому арифметические действия над предопределенными типами, присваивание предопределенных типов (особенно указателей) и другие низкоуровневые операции не приведут к возникновению исключения. (В результате их выполнения мог бы возникнуть сигнал или прерывание некоторого рода, но это не исключения.)

Перегрузка операторов и шаблоны усложняют ситуацию, поскольку часто трудно определить, приведет ли выполнение той или иной операции к вызову функции и потенциальному исключению. Например, я знаю,

[1] Речь идет о надежности в случае возникновения исключительной ситуации (Exception safety). – *Примеч. перев.*

что исключение не возникнет, если происходит присваивание символьных указателей, тогда как в случае присваивания пользовательских строк (String) это возможно:

```
const char *a, *b;
String c, d;
//...
a = b;   // нет вызова функции, нет исключения
c = d;   // вызов функции, возможно исключение
```

В случае применения шаблонов неопределенность возрастает:

```
template <typename T>
void aTemplateContext() {
   T e, f;
   T *g, *h;
   //...
   e = f;   // вызов функции? исключение?
   g = h;   // нет вызова функции, нет исключения
   //...
}
```

Из-за этой неопределенности необходимо принять, что все потенциальные вызовы функций в рамках шаблона представляют собой вызовы функций. Это касается инфиксных операторов, неявных преобразований и т. д.

Аксиома 2: уничтожать безопасно

Эта аксиома скорее социальная, чем техническая. Условно деструкторы, operator delete и operator delete[], не формируют исключений. Рассмотрим воображаемый деструктор, который должен удалять два указателя на данные-члены. Мы знаем, что подвергнемся критике, гонениям и изоляции, если позволим исключению сформироваться в деструкторе, поэтому задействуем блок try:

```
X::~X() {
   try {
      delete ptr1_;
      delete ptr2_;
   }
   catch( ... ) {}
}
```

Этот стиль отнюдь не обязателен, но угроза неодобрения окружающих (надеемся и рассчитываем на это) влияет на авторов деструкторов и функций operator delete объектов, на которые ссылаются ptr1_ и ptr2_. Эти страхи тоже могут быть нам полезными и упростить задачу:

```
X::~X() {
   delete ptr1_;
   delete ptr2_;
}
```

Аксиома 3: обмен значениями не формирует исключение

Есть еще одна сформированная общественным мнением аксиома, но она не настолько прочно укоренилась, как запрет формирования исключений в деструкторах и при удалении. Обмен значениями – не очень распространенная операция, но она широко применяется «за кулисами», особенно в реализациях STL. Обмен значениями происходит при любой сортировке, разбиении на разделы и множестве других операций. И надежный обмен значениями очень важен для надежности этих операций. (См. также тему 13 «*Операции копирования*».)

Тема 39 | Надежные[1] функции

Основная трудность в написании надежного кода не в формировании или перехвате исключений, а в том, что происходит между указанными событиями. По мере того как сформированное исключение прокладывает свой путь от выражения `throw` к блоку `catch`, каждая частично выполненная функция на этом пути должна «очистить» все важные контролируемые ею ресурсы до того, как ее активационная запись будет вытолкнута из стека выполнения. Обычно (но не всегда) для написания надежной функции необходимы только недолгие размышления и немного здравого смысла.

Например, рассмотрим реализацию присваивания `String` из темы 12 «*Присваивание и инициализация – это не одно и то же*»:

```
String &String::operator =( const char *str ) {
    if( !str ) str = "";
    char *tmp = strcpy( new char[ strlen(str)+1 ], str );
    delete [] s_;
    s_ = tmp;
    return *this;
}
```

Реализация этой функции может показаться чересчур витиеватой, поскольку ее можно было бы сделать более короткой и не использовать временную переменную:

```
String &String::operator =( const char *str ) {
    delete [] s_;
    if( !str ) str = "";
    s_ = strcpy( new char[ strlen(str)+1 ], str );
```

[1] Подразумевается надежность в смысле exception-safety. – *Примеч. перев.*

```
    return *this;
}
```

Однако если функция уничтожения массива ограничена общественным обязательством не формировать исключений (см. тему 38 «*Аксиомы надежности исключений*»), функция создания массива, располагающаяся несколькими строками ниже, не дает таких обещаний. Если удалить старый буфер до того, как станет известно об успешном выделении нового буфера, объект `String` останется в некорректном состоянии. Эта ситуация хорошо описана в книге [8]. Передам своими словами: сначала надо сделать что-то, что могло бы обусловить возникновение исключения без изменения значимого состояния, и затем использовать операции, которые не могут сформировать исключения до полного завершения их выполнения. Это то, что было сделано в первой приведенной выше реализации `String::operator =`. Давайте рассмотрим другой пример из темы 19 «*Команды и Голливуд*»:

```
void Button::setAction( const Action *newAction ) {
    Action *temp = newAction->clone(); // возможно возникновение исключения
                                       // без изменения значимого состояния...
    delete action_; // затем меняем состояние!
    action_ = temp;
}
```

Поскольку это виртуальная функция, нам фактически ничего не известно о поведении функции `clone` при возникновении исключения, поэтому предполагаем худшее. Если операция `clone` проходит успешно, дальше выполняются надежные удаление и присваивание указателя. В противном случае формирование исключения приведет к преждевременному выходу из `Button::setAction` без всякого вреда. Программисты на C++ никогда не пытаются «очищать» код таким образом, делая его ненадежным:

```
void Button::setAction( const Action *newAction ) {
    delete action_;                    // изменяет состояние!
    action_ = newAction->clone();      // затем может быть
                                        // сформирует исключение?
}
```

Если уничтожение (предположительно, безопасное) осуществляется перед клонированием (которое не дает таких гарантий), и операция клонирования формирует исключение, объект `Button` остается в неопределенном состоянии

Обратите внимание, что в правильно написанном надежном коде блоки `try` используются сравнительно мало. Пытаясь написать надежный код,

новички обильно снабжают его ненужными и подчас разрушительными блоками try и catch:

```
void Button::setAction( const Action *newAction ) {
    delete action_;
    try {
       action_ = newAction->clone();
    }
    catch( ... ) {
       action_ = 0;
       throw;
    }
}
```

Эта версия с ее аляповатыми блоками try и catch надежна в том смысле, что объект Button остается в непротиворечивом (но, вероятно, измененном) состоянии в случае, если clone формирует исключение. Однако наша предыдущая версия была короче, проще и надежней, потому что объект Button оставался не только непротиворечивым, но и неизмененным.

Лучше всего свести использование блоков try к минимуму и применять их главным образом там, где действительно необходимо проверить тип передаваемого исключения (чтобы что-то сделать с ним). На практике это обычно границы модулей между вашим кодом и библиотеками сторонних производителей и между вашим кодом и операционной системой.

Тема 40 | Методика RAII

Сообщество разработчиков на C++ имеет долгую и величественную традицию загадочных аббревиатур и странных названий методик. RAII объединяет в себе и странность, и загадочность. Она расшифровывается как «resource acquisition is initialization» – «выделение ресурса есть инициализация». (Нет, не «инициализация есть выделение ресурса», как вы могли подумать. Если хотите выглядеть странно, вы должны пройти весь путь от начала до конца, иначе все пропало.)

RAII – простая методика, использующая C++-понятие времени жизни объекта для управления такими ресурсами программ, как память, дескрипторы файлов, сетевые соединения, журнал аудита и т. д. Базовая техника проста. Если необходимо отследить важный ресурс, создается объект и время жизни ресурса ассоциируется со временем жизни объекта. Так, для управления ресурсами могут использоваться мудреные возможности управления объектами C++. Самая простая форма – создается объект, конструктор которого захватывает, а деструктор – высвобождает ресурс.

```
class Resource { ... };
class ResourceHandle {
   public:
      explicit ResourceHandle( Resource *aResource )
         : r_(aResource) {} // захват ресурса
      ~ResourceHandle()
         { delete r_; }     // высвобождение ресурса
      Resource *get()
         { return r_; }     // доступ к ресурсу
   private:
      ResourceHandle( const ResourceHandle & );
      ResourceHandle &operator =( const ResourceHandle & );
      Resource *r_;
};
```

В объекте ResourceHandle любопытно то, что если он объявлен как локальная переменная функции, аргумент функции или статический объект, мы гарантированно получаем вызов деструктора, и ресурс восстанавливается. Это свойство заслуживает особого внимания, если требуется отслеживать важные ресурсы, а сопровождение кода оставляет желать лучшего или нет спасения от исключений. Рассмотрим простой код, который не использует RAII:

```
void f() {
   Resource *rh = new Resource;
   //...
   if( iFeelLikeIt() ) // результат неудачного сопровождения
      return;
   //...
   g();                // исключение?
   delete rh;          // мы всегда сюда попадаем?
}
```

Исходная версия этой функции может быть надежной и всегда восстанавливать ресурс, на который ссылается rh. Однако со временем такой код может начать давать сбои, поскольку менее опытные разработчики вводят ранние возвраты, вызовы функций, которые могут формировать исключения или любым другим способом обходить код восстановления ресурса в конце функции. При использовании RAII функция становится и проще, и надежнее:

```
void f() {
   ResourceHandle rh( new Resource );
   //...
   if( iFeelLikeIt() ) // no problem!
      return;
   //...
   g();                // исключение? никаких проблем!
   // деструктор rh осуществляет уничтожение!
}
```

При использовании RAII вызов деструктора не гарантируется только в том случае, если дескриптор ресурса размещен в куче, потому что тогда деструктор вызывается лишь при явном удалении объекта. (Чтобы быть совершенно точным, необходимо упомянуть пограничные случаи, когда вызываются abort или exit, и неопределенные ситуации, которые могут иметь место, если сформированное исключение не перехватывается.)

```
ResourceHandle *rhp =
    new ResourceHandle(new Resource); // плохая идея!
```

RAII настолько распространена в программировании на C++, что трудно найти библиотечный компонент или значительный по объему блок кода, не использующий ее тем или иным образом. Обратите внимание, что определение «ресурса», который может управляться посредством RAII, очень широкое. Кроме ресурсов, фактически являющихся блоками памяти (буферы, строки, реализации контейнеров и т. д.), RAII можно использовать для управления такими системными ресурсами, как дескрипторы файлов, семафоры и сетевые соединения, а также менее эффектными сеансами регистрации, графическими формами или животными зоопарка.

Рассмотрим следующий класс:

```
class Trace {
   public:
      Trace( const char *msg ) : msg_(msg)
       { std::cout << "Entering " << msg_ << std::endl; }
      ~Trace()
       { std::cout << "Leaving " << msg_ << std::endl; }
   private:
      std::string msg_;
};
```

В случае с классом Trace контролируемым ресурсом является сообщение, которое должно распечатываться при выходе из области видимости. Полезно проследить поведение разных объектов Trace (автоматического, статического, локального и глобального), отслеживая их времена жизни в различных типах потока управления.

```
void f() {
   Trace tracer( "f" ); // распечатываем "входное" сообщение
   ResourceHandle rh( new Resource ); // захват ресурса
   //...
   if( iFeelLikeIt() )  // проблем нет!
      return;
   //...
   g();                 // исключение? никаких проблем!
   // деструктор rh осуществляет уничтожение!
   // деструктор tracer распечатывает сообщение выхода!
}
```

Приведенный выше код также иллюстрирует важный инвариант активации структуры конструктора и деструктора – активации из стека. То есть tracer объявляется и инициализируется до rh, благодаря чему rh будет гарантированно уничтожен раньше tracer (инициализируется последним, уничтожается первым). Говоря более обобщенно, когда объявляется последовательность объектов, эти объекты инициализируются во время выполнения в определенном порядке и уничтожаются в обратном порядке.

Этот порядок уничтожения не изменится даже в случае непредвиденного выхода, распространяющегося исключения, необычного `switch` или гибельного `goto`. (Если это утверждение вызывает сомнения, поработайте с классом `Trace`. Очень поучительно.) Это свойство особенно важно для захвата и высвобождения ресурсов, поскольку обычно ресурсы должны захватываться в определенном порядке и высвобождаться в обратном порядке. Например, сетевое соединение должно открываться перед отправкой контрольного сообщения, а закрывающее контрольное сообщение должно отправляться перед закрытием соединения.

Такое поведение, основанное на структуре стека, распространяется даже на инициализацию и уничтожение отдельных объектов. Конструктор объекта инициализирует сначала подобъекты своего базового класса в порядке их объявления, а затем данные-члены в порядке их объявления. После этого (и только тогда) выполняется тело конструктора. Теперь нам известно, как будет вести себя деструктор объекта. Проследуем в обратном порядке. Сначала выполняется тело деструктора, затем уничтожаются данные-члены в порядке их объявления и наконец уничтожаются подобъекты базового класса объекта в порядке обратном порядку их объявления. Когда порядок выполнения не так очевиден, стекоподобное поведение оказывается удобным для захвата и высвобождения необходимых объекту ресурсов.

Тема 41 | Операторы new, конструкторы и исключения

Чтобы написать идеальный надежный код, необходимо отслеживать все выделенные ресурсы и быть готовым высвободить их в случае возникновения исключения. Обычно это делается просто. Можно либо организовать код таким образом, чтобы восстановления ресурсов не требовалось (см. тему 39 «*Надежные функции*»), либо использовать дескрипторы ресурсов для автоматического высвобождения ресурсов (см. тему 40 «*Методика RAII*»). В экстремальных ситуациях можно отставить все условности и использовать блоки `try` или даже вложенные блоки `try`, но это должно быть исключением, а не правилом.

Однако существует несомненная проблема с использованием оператора `new`. На самом деле оператор `new` осуществляет две отдельные операции (см. тему 35 «*Синтаксис размещения new*»). Сначала он вызывает функцию `operator new` для выделения некоторой области памяти, а затем может инициировать конструктор, чтобы превратить это неинициализированное хранилище в объект:

```
String *title = new String( "Kicks" );
```

Проблема состоит в том, что если возникает исключение, невозможно определить, было ли оно сформировано функцией `operator new` или конструктором `String`. Это важно, потому что если выполнение `operator new` прошло успешно и исключение сформировал конструктор, вероятно, необходимо вызвать `operator delete` для выделенного (но неинициализированного) хранилища. Если исключение сформировала функция `operator new`, то память не была выделена и нет необходимости вызывать `operator delete`.

Можно вручную обеспечить необходимое поведение, разделив выделение и инициализацию и втиснув блок `try`, но это один из самых ужасных вариантов:

```
String *title // распределение памяти
   = static_cast<String *>(::operator new(sizeof(String));
try {
   new( title ) String( "Kicks" ); // синтаксис размещения new
}
catch( ... ) {
   ::operator delete( title ); // очистка ресурсов, если конструктор
                               // формирует исключение
}
```

Ой-ой-ой! В этом коде столько ошибок, что данный подход мы не будем даже рассматривать. Кроме того, что он добавляет хлопот разработчику, и без того заваленному работой, приведенный код не будет вести себя правильно, если у `String` будут члены `operator new` и `operator delete` (см. тему 36 «*Индивидуальное управление памятью*»). Это идеальный пример заумного кода, который сначала работает, но в будущем дает скрытый сбой из-за незначительных изменений (например, если кто-то вводит индивидуальное для `String` управление памятью).

К счастью, компилятор обрабатывает эту ситуацию и создает код, работающий так же, как приведенный выше «рукотворный» подход, но за одним маленьким исключением. Он будет вызывать функцию `operator delete`, соответствующую функции `operator new`, используемой для выделения памяти.

```
String *title = new String( "Kicks" );  // использовать члены, если есть
String *title = ::new String( "Kicks" ); // использовать глобальный new/delete
```

В частности, если память выделяется при помощи члена `operator new` и конструктор `String` формирует исключение, для высвобождения памяти будет вызван соответствующий член `operator delete`. Это еще одна веская причина объявить член `operator delete` при объявлении члена `operator new`.

В сущности то же самое происходит и при создании массивов, и при распределении памяти, когда используются перегруженные версии `operator new[]`. Компилятор будет пытаться найти и вызвать соответствующую `operator delete[]`.

Тема 42 | Умные указатели

Мы, программисты на C++, отличаемся особой верностью. Если нам нужна какая-то возможность, не поддерживаемая языком, мы не предаем C++ и не заигрываем с другими языками программирования. Мы просто расширяем C++, чтобы он поддерживал ту возможность, которая нам необходима.

Например, часто требуется что-то, ведущее себя как указатель. При этом встроенный тип указателя не годится. В таких случаях программист на C++ будет использовать *умный указатель*. (Аналогичные соображения по поводу указателей на функции ищите в теме 18 «*Объекты-функции*».)

Умный указатель – это тип класса, искусно замаскированный под указатель, но предоставляющий дополнительные возможности, отсутствующие у встроенного указателя. Как правило, умный указатель использует возможности, предоставляемые конструкторами, деструктором и операциями копирования класса, для управления доступом или отслеживания того, на что он указывает, таким способом, который недоступен встроенному указателю.

Все умные указатели перегружают операторы `->` и `*`, чтобы их можно было применять со стандартным синтаксисом указателей. (Иногда дело доходит до того, что перегружается оператор `->*`; см. тему 15 «*Указатели на члены класса – это не указатели*».) Другие умные указатели (в частности, используемые как итераторы STL) перегружают другие операторы указателей, такие как `++`, `--`, `+`, `-`, `+=`, `-=` и `[]` (см. тему 44 «*Арифметика указателей*»). Умные указатели часто реализуются как шаблоны классов, чтобы ссылаться на разные типы объектов. Вот очень простой шаблон умного указателя, проверяющий перед использованием, не является ли он нулевым:

```
template <typename T>
```

```
class CheckedPtr {
   public:
     explicit CheckedPtr( T *p ) : p_( p ) {}
      ~CheckedPtr() { delete p_; }
      T *operator ->() { return get(); }
      T &operator *() { return *get(); }
   private:
      T *p_;     // на что мы указываем
      T *get() { // проверяем указатель, перед тем как вернуть его
         if( !p_ )
            throw NullCheckedPointer();
         return p_;
      }
      CheckedPtr( const CheckedPtr & );
      CheckedPtr &operator =( const CheckedPtr & );
};
```

Умный указатель должен имитировать встроенный и быть понятным:

```
CheckedPtr<Shape> s( new Circle );
s->draw();      // то же, что и (s.operator ->())->draw()
```

Ключом к этому фасаду является перегруженный `operator ->`. Оператор `->` должен быть перегружен как член и обладать довольно необычным свойством: его можно не указывать при вызове. Иначе говоря, когда написано `s->draw()`, компилятор понимает, что `s` не указатель, а объект класса с перегруженным `operator ->` (то есть `s` – умный указатель). В результате вызывается перегруженный оператор-член, который возвращает (в данном случае) встроенный указатель `Shape *`. Затем этот указатель используется для вызова функции `draw` объекта `Shape`. Если написать это без сокращений, получится запутанное выражение `(s.operator ->())->draw()`, дважды использующее оператор `->`, один перегруженный и один встроенный.

Умные указатели обычно перегружают также оператор `*`, как и оператор `->`, чтобы их можно было использовать для типов, не являющихся классами.

```
CheckedPtr<int> ip = new int;
*ip = 12;     // аналогично ip.operator *() = 12
(*s).draw();  // используется также для указателя на класс
```

Умные указатели широко применяются в программировании на C++, начиная от дескрипторов ресурсов (см. темы 40 «*Методика RAII*» и 43 «*Указатель auto_ptr – штука странная*») и заканчивая итераторами STL, обертками указателей на функции-члены и т. д. и т. п. *Semper fidelis*[1].

[1] «Всегда верен» (*лат.*) – девиз корпуса морской пехоты. – *Примеч. перев.*

Тема 43 | Указатель auto_ptr – штука странная

Обсуждая RAII, надо обязательно упомянуть auto_ptr.[1] Как отмечалось в теме 40 «*Методика RAII*», дескрипторы ресурсов широко применяются в программировании на C++. Поэтому стандартная библиотека поставляет шаблон дескриптора ресурсов, auto_ptr, удовлетворяющий большинству потребностей в дескрипторе ресурсов. Шаблон класса auto_ptr служит для создания умных указателей (см. тему 42 «*Умные указатели*»), которые знают, как очищать ресурсы после себя.

```
using std::auto_ptr;  // см. тему 23 «Пространства имен»
auto_ptr<Shape> aShape( new Circle );
aShape->draw();       // рисуем круг
(*aShape).draw();     // рисуем еще раз
```

Как все хорошо спроектированные умные указатели, auto_ptr перегружает операторы -> и *, чтобы всегда можно было сделать вид, что работаешь со встроенным указателем. Указатель auto_ptr обладает многими замечательными свойствами. Во-первых, он очень эффективен. Вы никогда не добьетесь лучшей производительности, написав свой код со встроенным указателем. Во-вторых, когда auto_ptr выходит из области видимости, его деструктор высвободит все, на что бы он ни указывал, точно так же, как это сделал бы написанный вручную дескриптор указателя. В приведенном выше фрагменте кода объект Circle, на который ссылается aShape, будет уничтожен.

Третье замечательное свойство auto_ptr – при преобразованиях он ведет себя как встроенный указатель:

```
auto_ptr<Circle> aCircle( new Circle );
aShape = aCircle;
```

---

[1] В настоящее время auto_ptr объявлен устаревшим. Стандарт ISO/IEC 14882:2011 рекомендует использовать unique_ptr. – *Примеч. науч. ред.*

При разумном использовании шаблонных функций-членов (см. тему 50 «*Члены-шаблоны*») один auto_ptr может быть скопирован в другой, если это возможно для соответствующих встроенных указателей. В приведенном выше коде auto_ptr<Circle> может быть присвоен auto_ptr<Shape>, потому что Circle * может быть присвоен Shape * (предполагается, что Shape — это открытый базовый класс Circle).

В чем auto_ptr отличается от обычного умного указателя (или даже обычного объекта), так это в операциях копирования. Для обычного класса операции копирования (см. тему 13 «*Операции копирования*») не влияют на источник копирования. Иначе говоря, если T — это некоторый тип,

```
T a;
T b( a );  // копирующее создание b значением a
a = b;     // присваивание из b к a
```

то при инициализации b значением a значение a остается нетронутым, и когда b присваивается a, значение b остается нетронутым. Но с auto_ptr все не так! При присваивании aCircle aShape, которое мы видим выше, затрагиваются и источник, и цель присваивания. Если aShape был ненулевым, все, на что он ссылался, удаляется и заменяется тем, на что указывает aCircle. Кроме того, aCircle становится нулевым. Присваивание и инициализация auto_ptr — это не совсем операции копирования; это операции, которые передают управление базовым объектом от одного auto_ptr другому. Правый аргумент присваивания можно рассматривать как «источник», а левый — как «приемник». Управление базовым объектом передается от источника к приемнику. Это полезное свойство для работы с ресурсами.

Однако следует избегать применения auto_ptr в двух распространенных ситуациях. Во-первых, они никогда не должны выступать в качестве элементов контейнера. Часто эти элементы копируются в рамках контейнера, и контейнер будет считать, что его элементы подчиняются обычной семантике копирования, не задействующей auto_ptr,. Умный указатель может применяться как элемент контейнера просто потому, что он не auto_ptr. Во-вторых, auto_ptr должен ссылаться на один элемент, но не на массив. Причина в том, что при уничтожении объекта, на который ссылается auto_ptr, будет вызвана operator delete, а не функция уничтожения массива. Если auto_ptr ссылается на массив, будет вызван несоответствующий оператор уничтожения (см. тему 37 «*Создание массивов*»).

```
vector< auto_ptr<Shape> > shapes;   // скорее всего ошибка, неудачная идея
auto_ptr<int> ints( new int[32] );  // неудачная идея, ошибки нет (пока)
```

В общем стандартные vector или string представляют собой разумную альтернативу auto_ptr для массива.

Тема 44 | Арифметика указателей

Арифметика указателей проста. Чтобы понять ее принцип в C++, лучше всего рассмотреть указатель на массив:

```
const int MAX = 10;
short points[MAX];
short *curPoint = points+4;
```

Это дает нам массив и указатель примерно на середину массива (рис. 44.1).

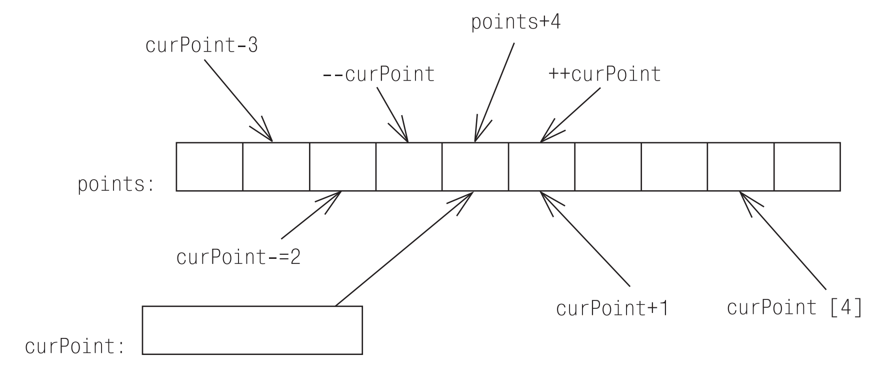


Рис. 44.1. *Результат различных арифметических операций над адресом, содержащимся в указателе*

Давая положительное или отрицательное приращение указателю `curPoint`, мы перемещаем его на следующий или предыдущий элемент `short` массива `points`. Другими словами, арифметика указателей всегда масштабируется соответственно размеру объекта, на который они указывают. Приращение `curPoint` на единицу не добавляет один байт к адресу указателя, до-

бавляется sizeof(short) байт. Вот почему арифметика указателей не может применяться к указателям void *: неизвестно, на объект какого типа ссылается void *, поэтому невозможно правильно отмасштабировать операции.

Единственный случай, когда эта простая схема, кажется, приводит к путанице, – это многомерные массивы. Неопытные программисты на C++ забывают, что многомерный массив – это массив массивов:

```
const int ROWS = 2;
const int COLS = 3;
int table[ROWS][COLS];       // массив ROWS массивов, каждый из которых содержит
                            // COLS элементов типа int
int (*ptable)[COLS] = table;  // указатель на массив из COLS элементов типа int
```

Двумерный массив, показанный на рис. 44.2, удобно представить как таблицу, несмотря на то, что на самом деле в памяти он располагается линейно (рис. 44.3).

table:

0,0	0,1	0,2
1,0	1,1	1,2

Рис. 44.2. *Двумерный массив концептуально является таблицей*

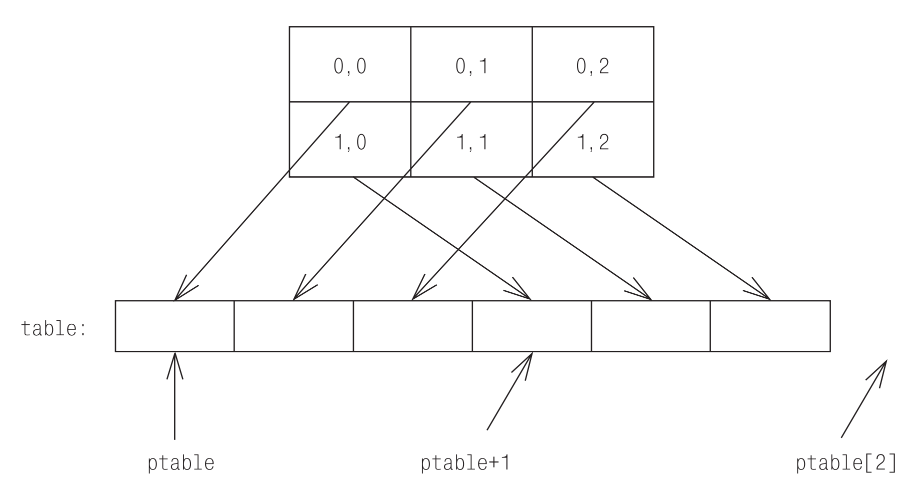


Рис. 44.3. *На самом деле двумерный массив является линейной последовательностью одномерных массивов*

В случае применения арифметики указателей к `ptable` приращения, как всегда, масштабируются соответственно размеру объекта, на который указывает `ptable`. Но этот объект имеет тип массива из `COLS` целочисленных элементов (его размер равен `sizeof(int)*COLS` байт), а не `int`.

Над указателями одного типа может быть проведена операция вычитания. В результате получаем количество объектов (а не байтов), находящихся между этими двумя указателями. Если первый указатель больше (указывает на область памяти с большим адресом), чем второй, то результат будет положительный; в противном случае – отрицательный. Если два указателя ссылаются на один и тот же объект или оба являются нулевыми, в результате будет получен нуль. Тип результата вычитания двух указателей – стандартное переименование типа `ptrdiff_t`, который обычно является псевдонимом для `int`. Два указателя нельзя подвергнуть операциям сложения, умножения или деления, потому что они просто не имеют смысла для адресов. Указатели – это не целые числа (см. тему 35 «*Синтаксис размещения new*»).

Эта обычно понятная концепция арифметики указателей используется в качестве модели при проектировании итераторов STL (см. темы 4 «*Стандартная библиотека шаблонов*» и 42 «*Умные указатели*»). Итераторы STL также допускают указателеподобную арифметику, в которой используется синтаксис, аналогичный встроенным указателям. По сути, встроенные указатели представляют собой совместимые итераторы STL. Рассмотрим возможную реализацию контейнера STL «список» (рис. 44.4).

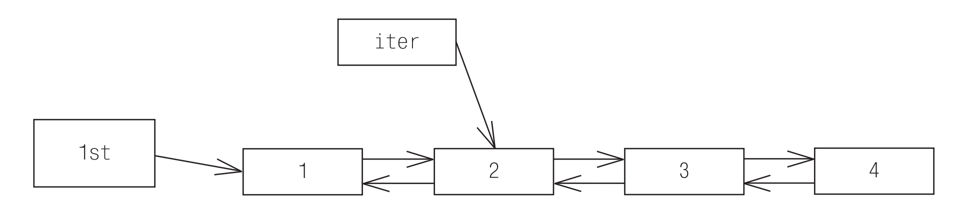


Рис. 44.4. *Возможная реализация стандартного списка. Итератор списка не является указателем, но он смоделирован как указатель*

Такая конфигурация могла бы возникнуть в результате выполнения следующего кода:

```
int a[] = { 1, 2, 3, 4 };
std::list<int> lst(a, a+4);
std::list<int>::iterator iter = lst.begin();
++iter;
```

Итератор списка не может быть встроенным указателем, а является умным указателем с перегруженными операторами. Операция `++iter` указателеподобной арифметики не дает приращения `iter` так, как это было бы при приращении указателя. Она перемещает ссылку от текущего узла списка к следующему. Однако аналогия с арифметикой для встроенных указателей абсолютная: операция приращения перемещает итератор к следующему элементу списка, так же как приращение встроенного указателя перемещает его к следующему элементу массива.

Тема 45 | Терминология шаблонов

Строго соблюдать терминологию важно в любой технической области, особенно в программировании, еще важнее в программировании на C++ и исключительно важно в программировании на C++ с использованием шаблонов.

Наиболее важные аспекты терминологии шаблонов C++ иллюстрирует рис. 45.1.

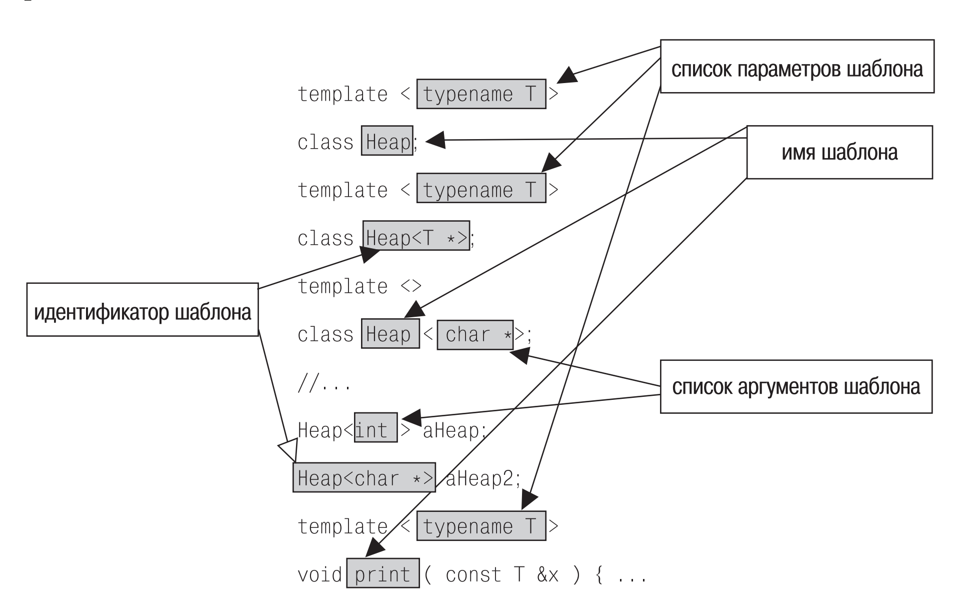


Рис. 45.1. *Терминология шаблонов. Строгое соблюдение терминологии исключительно важно для точности передачи конструкции шаблона*

Особенно четко надо различать параметр шаблона, который задается в объявлении шаблона, и аргумент шаблона, указываемый в специализации шаблона.

```
template <typename T> // T - параметр шаблона
class Heap { ... };
//...
Heap<double> dHeap; // double - аргумент шаблона
```

Кроме того, надо четко различать имя шаблона, которое является простым идентификатором, и идентификатор шаблона, представляющий собой имя шаблона со списком аргументов шаблона.

Большинство программистов на C++ путают «создание экземпляра» и «специализацию». Специализация шаблона – это то, что получается, если снабдить шаблон набором аргументов шаблона. Специализация может быть явной или неявной. Например, записывая `Heap<int>`, мы явно специализируем класс `Heap` аргументом `int`. При записи `print( 12.3 )` имеет место неявная специализация шаблона функции `print` аргументом `double`. Специализация шаблона может приводить к созданию экземпляра шаблона. Так, если доступна версия `Heap` для `int`, специализация `Heap<int>` будет ссылаться на эту версию, экземпляр не будет создаваться (см. тему 46 «*Явная специализация шаблона класса*»). Однако если используется *базовый*[1] шаблон `Heap` или частичная специализация (см. тему 47 «*Частичная специализация шаблонов*»), экземпляр будет создаваться.

---

[1] Базовым шаблоном называется общий случай вида:

```
template<typename T1, typename T2, … , typename TN> MyClass { … };
```

Подробнее см. тему 46. – *Примеч. науч. ред.*

Тема 46 | Явная специализация шаблона класса

Осуществить явную специализацию шаблона класса легко. Во-первых, необходим общий случай, который требуется специализировать. Этот общий случай называют *базовым* шаблоном.

```
template <typename T> class Heap;
```

Чтобы провести специализацию базового шаблона, его достаточно объявить (как приведенный выше `Heap`), но обычно его еще и определяют (как `Heap` ниже):

```
template <typename T>
class Heap {
   public:
      void push( const T &val );
      T pop();
      bool empty() const { return h_.empty(); }
   private:
      std::vector<T> h_;
};
```

Наш базовый шаблон реализует структуру данных кучи, скрывая запутанные алгоритмы работы с кучей за удобным в работе интерфейсом. Куча – это линеаризованная структура типа дерево, оптимизированная для ввода и извлечения данных. Операция вставки в кучу есть вставка элемента в древоподобную структуру. Операция извлечения из кучи есть удаление наибольшего элемента этой кучи и возврат его значения в качестве результата. Например, операции `push` и `pop` могут реализовываться с помощью стандартных алгоритмов `push_heap` и `pop_heap`:

```
template <typename T>
void Heap<T>::push( const T &val ) {
   h_.push_back(val);
   std::push_heap( h_.begin(), h_.end() );
}

template <typename T>
T Heap<T>::pop() {
   std::pop_heap( h_.begin(), h_.end() );
   T tmp( h_.back() );
   h_.pop_back();
   return tmp;
}
```

Данная реализация хорошо работает для многих типов значений, но дает сбой для указателей на символы. По умолчанию стандартные алгоритмы работы с кучей для сравнения и организации элементов кучи задействуют оператор <. Однако в случае с указателями на символы это привело бы к тому, что куча организовывалась бы по адресам строк, на которые ссылаются указатели символов, а не по самим значениям строк. Другими словами, по значениям указателей, а не по значениям того, на что они указывают.

Решить эту проблему можно путем явной специализации базового шаблона Heap для указателей на символы:

```
template <>
class Heap<const char *> {
   public:
      void push( const char *pval );
      const char *pop();
      bool empty() const { return h_.empty(); }
   private:
      std::vector<const char *> h_;
};
```

Список параметров шаблона пуст, но аргумент шаблона, для которого создается специализация, добавлен к имени шаблона. Любопытно, что эта явная специализация шаблона класса не есть шаблон, потому что в ней не осталось неопределенных параметров шаблона. Поэтому явную специализацию шаблона класса обычно называют *полной специализацией*, чтобы отличить ее от частичной специализации, которая является шаблоном (см. тему 47 «*Частичная специализация шаблонов*»).

Терминология в данной области несколько запутанная. Специализация шаблона – это имя шаблона с предоставленными аргументами шаблона (см. тему 45 «*Терминология шаблонов*»). Синтаксис Heap<const char *> – это специализация шаблона, как и Heap<int>. Однако первая специализа-

ция не приведет к созданию экземпляра шаблона Heap (потому что будет использоваться явная специализация, определенная для const char *). А вот вторая специализация обусловит создание экземпляра базового шаблона Heap.

Реализация специализации может быть настроена соответственно нуждам типа элемента const char *. Например, операция push может вставлять в кучу новое значение на основании значения строки, на которую ссылается указатель, а не адреса, содержащегося в указателе:

```
bool strLess( const char *a, const char *b )
   { return strcmp( a, b ) < 0; }

void Heap<const char *>::push( const char *pval ) {
   h_.push_back(pval);
   std::push_heap( h_.begin(), h_.end(), strLess );
}
```

Обратите внимание на отсутствие ключевого слова template («шаблон») и списка параметров в определении Heap<const char *>::push. Это не шаблон функции, потому что, как отмечалось ранее, явная специализация Heap<const char *> — это не шаблон.

Имея в распоряжении эту полную специализацию, мы можем отличать Heap для типа const char * от прочих Heap:

```
Heap<int> h1;          // используется базовый шаблон
Heap<const char *> h2; // используется явная специализация
Heap<char *> h3;       // используется базовый шаблон!
```

Компилятор сравнивает специализацию шаблона класса с объявлением базового шаблона. Если аргументы шаблона соответствуют базовому шаблону (в случае с Heap, если имеется единственный аргумент, указывающий имя типа), компилятор будет искать явную специализацию, точно соответствующую аргументам шаблона. Если мы хотим специализировать Heap для типа char * в дополнение к Heap для типа const char *, необходимо предоставить дополнительную явную специализацию.

```
template <>
class Heap<char *> {
   public:
      void push( char *pval );
      char *pop();
      size_t size() const;
      void capitalize();
      // метод empty() отсутствует!
   private:
      std::vector<char *> h_;
```

```
};
```

Обратите внимание на отсутствие требования обязательного совпадения между интерфейсами явной специализации и базового шаблона. Например, в первой явной специализации Heap для const char * тип формального аргумента функции push был объявлен как const char *, а не const char *&. Это разумная оптимизация для аргумента указателя. В специализации Heap для char * интерфейс еще больше отличается от интерфейса базового шаблона.

Добавлены две новые функции – size (вычисление размера) и capitalize (преобразование из строчных в заглавные), обе допустимые и иногда полезные, и исключена другая – empty (пустой), что допустимо, но в общем не рекомендуется. Обдумывая интерфейсы явных специализаций шаблонов классов, полезно провести аналогию с отношением между базовым и производными классами (хотя явная специализация шаблона класса не имеет абсолютно никакой связи с производными классами). Пользователи иерархии класса часто пишут в интерфейсе базового класса полиморфный код, предполагая, что производный класс реализует этот интерфейс (см. тему 2 «*Полиморфизм*»). Аналогично, в интерфейсе, предоставляемом в базовом шаблоне, обычно располагается универсальный код (если базовый шаблон объявляется и определяется) и предполагается, что любая специализация будет, по крайней мере, иметь такие возможности (хотя, как и в производных классах, в ней могут быть и дополнительные возможности). Рассмотрим простой шаблон функции:

```
template <typename T, typename Out>
void extractHeap( Heap<T> &h, Out dest ) {
   while( !h.empty() )
       *dest++ = h.pop();
}
```

Автор этого шаблона функции плохо подумает об авторе явной специализации Heap для char *, если данный код будет работать:

```
Heap<const char *> heap1;
//...
vector<const char *> vec1;
extractHeap( heap1, back_inserter(vec1) );  // хорошо...
```

а этот код не скомпилируется:

```
Heap<char *> heap2;
//...
vector<char *> vec2;
extractHeap( heap2, back_inserter(vec2) );  // ошибка! функция-член empty()
                          // отсутствует в явной специализации Heap для char *
```

Тема 47 | Частичная специализация шаблонов

Давайте уточним: нельзя частично специализировать шаблоны функций. Этой возможности просто нет в C++ (хотя возможно, что когда-нибудь она появится). Их можно перегрузить (см. тему 58 «*Перегрузка шаблонов функций*»). Так что мы будем рассматривать только шаблоны классов.

Принцип частичной специализации шаблона класса прост. Как и в случае с полной специализацией, сначала необходим общий случай, или базовый шаблон, который будет подвергнут специализации. Воспользуемся шаблоном `Heap` из темы 46 «*Явная специализация шаблона класса*»:

```
template <typename T> class Heap;
```

Явная специализация (в просторечии «полная специализация») используется, чтобы снабдить шаблон класса определенным набором аргументов. В теме 46 «*Явная специализация шаблона класса*» явная специализация применялась, чтобы предоставить специальные реализации `Heap` для `const char *` и `char *`. Однако нерешенной остается проблема с `Heap` для других типов указателей: хотелось бы упорядочивать `Heap` по значениям, на которые ссылаются элементы указателей, а не по значениям самих указателей.

```
Heap<double *> readings;   // базовый шаблон, T - это double *
```

Поскольку тип (`double *`) не соответствует ни одной из наших полных специализаций указателя на символ, компилятор создаст экземпляр базового шаблона. Можно было бы обеспечить полные специализации для `double *` и любого другого типа указателя, но это очень трудно и абсолютно непригодно к сопровождению. Это задача частичной специализации:

```
template <typename T>
```

```
class Heap<T *> {
   public:
      void push( const T *val );
      T *pop();
      bool empty() const { return h_.empty(); }
   private:
      std::vector<T *> h_;
};
```

Синтаксис частичной специализации подобен синтаксису полной специализации, но список параметров не пуст. Как в полной специализации, имя шаблона класса – это идентификатор шаблона, а не просто имя шаблона (см. тему 45 «*Терминология шаблонов*»).

Частичная специализация для указателей позволяет изменять реализацию. Например, вставки могут осуществляться на базе значения объекта, на который указывает указатель, а не значения указателя. Сначала давайте возьмем компаратор, сравнивающий два указателя по значениям, на которые они указывают (см. тему 20 «*Объекты-функции STL*»):

```
template <typename T>
struct PtrCmp : public std::binary_function<T *, T *, bool> {
   bool operator ()( const T *a, const T *b ) const
      { return *a < *b; }
};
```

Теперь применим этот компаратор для реализации необходимого поведения операции `push`:

```
template <typename T>
void Heap<T *>::push( T *pval ) {
   if( pval ) {
     h_.push_back(pval);
     std::push_heap( h_.begin(), h_.end(), PtrCmp<T>() );
   }
}
```

Обратите внимание, что, в отличие от полной специализации шаблона класса, частичная специализация представляет собой шаблон. Ключевое слово `template` и список параметров обязательны в определениях ее членов.

В отличие от приводимых выше полных специализаций, тип параметра этой версии `Heap` определен не полностью. Лишь частично определено, что это должен быть `T *`, где `T` – неопределенный тип. Именно поэтому данная специализация частичная. И ее приоритет при создании экземпляра `Heap` для любого (неопределенного) типа указателя будет выше, чем приоритет базового шаблона. Кроме того, полные специализации `Heap` для `const char *` и `char *` будут предпочтительнее этой частичной специализации, если типом аргумента шаблона будет `const char *` или `char *`.

```
Heap<std::string> h1;    // базовый, T является std::string
Heap<std::string *> h2;  // частичная специализация, T является std::string
Heap<int **> h3;         // частичная специализация, T является int *
Heap<char *> h4;         // полная специализация для char *
Heap<char **> h5;        // частичная специализация, T является char *
Heap<const int *> h6;    // частичная специализация, T является const int
Heap<int (*)()> h7;      // частичная специализация, T является int ()
```

Полный набор правил выбора частичной специализации из множества доступных довольно запутан, но в большинстве случаев делается это просто. Обычно выбирается самый специализированный, самый ограниченный вариант из возможных. Механизм частичной специализации точен и позволяет практически безошибочно выбирать необходимую версию. Например, можно было бы дополнить наш набор частичных специализаций вариантом с указателем на константу:

```
template <typename T>
class Heap<const T *> {
   //...
};
//...
Heap<const int *> h6; // другая частичная специализация, теперь T является int
```

Обратите внимание, что, как обсуждалось в теме 46 «*Явная специализация шаблона класса*», компилятор сравнивает специализацию шаблона класса с объявлением базового шаблона. Если аргументы шаблона совпадают с аргументами базового шаблона (в случае с `Heap` – если есть единственный аргумент имени типа), компилятор будет искать полную или частичную специализацию, более всего соответствующую аргументам шаблона.

Обратите внимание на такую тонкость: экземпляр полной или частичной специализации базового шаблона должен создаваться с таким же числом и типом аргументов, как и в базовом шаблоне, но форма списка параметров шаблона может отличаться от базового шаблона. В случае с `Heap` базовый шаблон принимает единственный параметр – имя типа. Таким образом, экземпляр любой полной или частичной специализации `Heap` должен создаваться с единственным аргументом – именем типа:

```
template <typename T> class Heap;
```

Следовательно, полная специализация `Heap` по-прежнему принимает единственный аргумент шаблона – имя типа, но список параметров шаблона отличается от списка базового шаблона, потому что он пуст:

```
template <> class Heap<char *>;
```

Частичная специализация Heap также должна принимать единственный аргумент шаблона – имя типа, и ее список параметров шаблона может содержать единственный параметр (имя типа) в заголовке шаблона:

```
template <typename T> class Heap<T *>;
```

но это не обязательно.

```
template <typename T, int n> class Heap<T [n]>;
```

Эта частичная специализация будет выбрана для специализации Heap для массива. Например:

```
Heap<float *[6]> h8; // частичная специализация, T является float *, и n равно 6
```

Эта частичная специализация гласит: «Данная частичная специализация принимает единственный параметр типа как базовый шаблон, но этот параметр должен иметь форму массива T размером n элементов». Рассмотрим несколько более сложных примеров:

```
template <typename R, typename A1, typename A2>
class Heap<R (*)(A1,A2)>;

template <class C, typename T>
class Heap<T C::*>;
```

С добавлением частичных специализаций появилась возможность обрабатывать случаи для Heap из указателей на функции-нечлены, принимающие два аргумента, и из указателей на данные-члены (хотя зачем могут понадобиться такие кучи, остается только догадываться):

```
Heap<char *(*)(int,int)> h9;    // частичная специализация
                                // R является char *, A1 и A2 являются int
Heap<std::string Name::*> h10;  // частичная специализация
                                // T является string, C является Name
```

Тема 48 | Специализация членов шаблона класса

По поводу явной и частичной специализаций шаблона класса широко распространено заблуждение, что специализация каким-то образом «наследует» что-то от базового шаблона. Это не так. Полная или частичная специализация шаблона класса – это совершенно самостоятельная сущность, которая не «наследует» ни интерфейс, ни реализацию базового шаблона. Однако если оставить в стороне технические вопросы, специализации действительно наследуют ряд ожиданий, касающихся интерфейсов и поведения. Поэтому пользователи, создающие универсальный код для интерфейса базового шаблона, обычно ожидают, что этот код будет работать и со специализациями.

Это подразумевает, что полная или частичная специализация должна в целом реализовать все возможности базового шаблона, даже если специализации требуется только часть реализации. Альтернативным решением часто является специализация лишь подмножества функций-членов базового шаблона. Рассмотрим базовый шаблон `Heap` (см. тему 46 «*Явная специализация шаблона класса*»):

```
template <typename T>
class Heap {
   public:
      void push( const T &val );
      T pop();
      bool empty() const { return h_.empty(); }
   private:
      std::vector<T> h_;
};
```

Наша полная специализация Heap для const char * заменила всю реализацию базового шаблона, даже несмотря на то, что его закрытая реализация и пустая функция-член идеально подходили для кучи указателей символов. Фактически надо было лишь специализировать функции-члены push и pop:

```
template <>
void Heap<const char *>::push( const char *const &pval ) {
   h_.push_back(pval);
   std::push_heap( h_.begin(), h_.end(), strLess );
}

template<>
const char *Heap<const char *>::pop() {
   std::pop_heap( h_.begin(), h_.end(), strLess );
   const char *tmp = h_.back();
   h_.pop_back(); return tmp;
}
```

Данные функции представляют собой явные специализации соответствующих членов базового шаблона Heap. Они будут использоваться вместо неявно созданных экземпляров специализаций Heap<const char *>.

Обратите внимание, что интерфейс каждой из этих функций должен точно совпадать с соответствующим интерфейсом шаблона, члены которого они специализируют. Например, в базовом шаблоне объявлено, что push принимает аргумент типа const T &. Таким образом, в явной специализации push для const char * должен быть аргумент типа const char *const &. (То есть ссылка на константный указатель на константный символ.) Заметьте, что этого ограничения не было, когда речь шла о полной специализации шаблона Heap в целом, где аргументом для push был объявлен просто const char *.

Увеличим уровень сложности (а это типично для программирования с применением шаблонов) и посмотрим, что произойдет, если в нашем распоряжении будет частичная специализация Heap для указателей в целом (см. тему 47 «*Частичная специализация шаблонов*»):

```
template <typename T>
class Heap<T *> {
   //...
   void push( T *pval );
   //...
};
```

Если эта частичная специализация Heap существует, то наша явная специализация push должна соответствовать интерфейсу члена push частичной специализации, поскольку в противном случае для Heap<const char *>

создавался бы экземпляр этой функции. Теперь явную специализацию необходимо объявить так:

```
template <>
void Heap<const char *>::push( const char *pval ) {
   h_.push_back(pval);
   std::push_heap( h_.begin(), h_.end(), strLess );
}
```

Два последних замечания. Во-первых, в дополнение к функциям-членам могут быть созданы явные специализации для других членов шаблонов класса, включая статические члены и члены-шаблоны.

Во-вторых, люди часто путают явную специализацию и явное создание экземпляра. Как было показано выше, явная специализация – это средство предоставления специальной версии шаблона или члена шаблона, которая отличается от того, что можно получить в результате неявного создания экземпляра. Явное создание экземпляра прямо указывает компилятору создать член, идентичный тому, который можно получить при неявном создании экземпляра.

```
template void Heap<double>::push( const double & );
```

Обратитесь также к теме 61 «*Мы создаем экземпляр того, что используем*».

Тема 49 | Устранение неоднозначности с помощью ключевого слова typename

Даже опытных разработчиков на C++ часто смущает слишком сложный синтаксис программирования с применением шаблонов. Из всех синтаксических кульбитов, которые приходится при этом выделывать, ни один так не сбивает с толку с самого начала, как необходимость помогать компилятору в устранении неоднозначности синтаксического анализа.

В качестве примера рассмотрим часть реализации шаблона простого нестандартного контейнера.

```
template <typename T>
class PtrList {
   public:
      //...
      typedef T *ElemT;
      void insert( ElemT );
      //...
};
```

Распространенная практика – встраивание информации о шаблоне посредством вложенных имен типов. Благодаря этому становится возможным получить информацию об экземпляре шаблона через соответствующее вложенное имя (см. темы 53 «*Встроенная информация о типе*» и 54 «*Свойства*»).

```
typedef PtrList<State> StateList;
//...
StateList::ElemT currentState = 0;
```

Вложенное имя ElemT обеспечивает возможность простого доступа к тому, что шаблон PtrList считает типом своего элемента. И хотя при создании экземпляра PtrList использовалось имя типа State, типом элемента является State *. При других обстоятельствах PtrList мог бы быть реализован с помощью умных указателей. Кроме того, PtrList может менять свою реализацию в зависимости от свойств типа, используемого для создания ее экземпляра (см. тему 52 «*Создание специализации для получения информации о типе*»). Вложенные имена типов помогают изолировать пользователей PtrList от этих внутренних решений реализации.

Вот еще один нестандартный контейнер:

```
template <typename Etype>
class SCollection {
  public:
    //...
    typedef Etype ElemT;
    void insert( const Etype & );
    //...
};
```

Оказывается, что SCollection спроектирован в соответствии с теми же стандартами присваивания имен, что и PtrList, потому что он тоже определяет вложенное имя типа ElemT. Соблюдать установленное соглашение полезно, потому что (наряду с другими преимуществами) это позволяет создавать универсальные алгоритмы, работающие с целым рядом различных типов контейнеров. Например, можно было бы написать простой полезный алгоритм, заполняющий соответствующий шаблон содержимым массива элементов соответствующего типа:

```
template <class Cont>
void fill( Cont &c, Cont::ElemT a[], int len ) {  // ошибка!
  for( int i = 0; i < len; ++i )
    c.insert( a[i] );
}
```

К сожалению, получаем синтаксическую ошибку. Вложенное имя Cont::ElemT не распознается как имя типа! Беда в том, что в контексте шаблона fill компилятор не располагает достаточной информацией и не может определить, является ли вложенное имя ElemT именем типа. Стандарт гласит, что в подобных ситуациях вложенное имя не считается именем типа.

Если на первых порах вам это кажется несущественным, вы не одиноки. Посмотрим, однако, какая информация доступна компилятору в разных контекстах. Сначала рассмотрим ситуацию с нешаблонным классом:

```
class MyContainer {
```

```
    public:
        typedef State ElemT;
        //...
};
//...
MyContainer::ElemT *anElemPtr = 0;
```

Здесь явно никаких проблем. Компилятор может проверить содержимое класса `MyContainer` (Мой контейнер), удостовериться в наличии в нем члена `ElemT` и отметить, что `MyContainer::ElemT` на самом деле представляет собой имя типа. Все так же просто, как и для класса, сгенерированного из шаблона класса.

```
typedef PtrList<State> StateList;
//...
StateList::ElemT aState = 0;
PtrList<State>::ElemT anotherState = 0;
```

Для компилятора созданный экземпляр шаблона класса является всего лишь классом, и нет никакой разницы в доступе к вложенному имени из класса `PtrList<State>` и из `MyContainer`. В любом случае компилятор просто проверяет содержимое класса, чтобы определить, является ли `ElemT` именем типа.

Однако в контексте шаблона все по-другому, потому что здесь доступно меньшее количество точной информации. Рассмотрим такой фрагмент:

```
template <typename T>
void aFuncTemplate( T &arg ) {
    ...T::ElemT...
```

Что известно компилятору при встрече с квалифицированным именем `T::ElemT`? Из списка параметров шаблона он знает, что `T` — это какое-то имя типа. Он также может установить, что `T` — это имя класса, потому что для доступа к вложенному имени `T` применен оператор разрешения области видимости (`::`). Но это все, что знает компилятор, потому что о содержимом `T` нет никакой доступной информации. Например, можно было бы вызвать `aFuncTemplate` с `PtrList`. В этом случае `T::ElemT` было бы именем типа.

```
PtrList<State> states;
//...
aFuncTemplate( states );  // T::ElemT является PtrList<State>::ElemT
```

Но что если бы надо было создать экземпляр `aFuncTemplate` другого типа?

```
struct X {
    enum Types { typeA, typeB, typeC } ElemT;
    //...
};
```

```
X anX;
//...
aFuncTemplate( anX );     // T::ElemT является X::ElemT
```

Здесь T::ElemT — это имя данных-члена, а не имя типа. Что делать компилятору? Компилятор бросает монетку и, если не может определить тип вложенного имени, полагает, что вложенное имя является именем нетипа. Что и обусловило синтаксическую ошибку в приведенном выше шаблоне функции fill.

Чтобы справиться с этой ситуацией, иногда необходимо явно сообщить компилятору о том, что вложенное имя является именем типа.

```
template <typename T>
void aFuncTemplate( T &arg ) {
   ...typename T::ElemT...
```

Здесь с помощью ключевого слова typename (имя типа) компилятору явно сообщается, что следующее полное имя является именем типа. Это дает возможность компилятору правильно проводить синтаксический анализ шаблона. Обратите внимание, мы сообщаем компилятору, что имя типа — это ElemT, а не T. Он уже может определить, что T — это имя типа. Аналогично следующая запись

```
typename A::B::C::D::E
```

сообщает компилятору о том, что (очень глубоко) вложенное имя E представляет собой имя типа.

Конечно, если создается экземпляр aFuncTemplate типа, не удовлетворяющего требованиям синтаксического анализа шаблона, это приведет к ошибке компиляции.

```
struct Z {
   // нет члена с именем ElemT...
};
Z aZ;
//...
aFuncTemplate( aZ );      // ошибка! нет члена Z::ElemT
aFuncTemplate( anX );     // ошибка! X::ElemT не является именем типа
aFuncTemplate( states );  // OK. вложенный ElemT является типом
```

Теперь можно переписать шаблон функции fill, чтобы синтаксический разбор проводился правильно:

```
template <class Cont>
void fill( Cont &c, typename Cont::ElemT a[], int len ) { // OK
   for( int i = 0; i < len; ++i )
      c.insert( a[i] );
}
```

Тема 50 | Члены-шаблоны

В шаблонах классов есть члены, сами по себе не являющиеся шаблонами. Многие из этих членов могут быть определены вне класса. Рассмотрим контейнер, реализующий однонаправленный список:

```
template <typename T>
class SList {
   public:
      SList() : head_(0) {}
      //...
      void push_front( const T &val );
      void pop_front();
      T front() const;
      void reverse();
      bool empty() const;
   private:
      struct Node {
         Node *next_;
         T el_;
      };
      Node *head_; // -> список
};
```

Когда функции-члены шаблона определяются вне шаблона класса, они имеют заголовок шаблона с такой же структурой, какая используется в описании шаблона класса:

```
template <typename T>
bool SList<T>::empty() const
    { return head_ == 0; }
```

Мы решили реализовать однонаправленный список как указатель на последовательность узлов, где каждый узел содержит элемент списка и ука-

затель на следующий узел списка. (При более сложной реализации усеченный Node, а не указатель на Node, мог бы встраиваться в SList, но для наших нужд этого достаточно.) Обычно такой вложенный тип класса определяется в самом шаблоне, но это необязательно:

```
template <typename T>
class SList {
  public:
    //...
  private:
    struct Node;    // неполное объявление класса
    Node *head_;    // -> список
    //...
};

template <typename T> // описание вне шаблона
struct SList<T>::Node {
    Node *next_;
    T el_;
};
```

Члены empty и Node — примеры членов шаблонов. Но шаблон класса (или даже нешаблонный класс) может иметь члены-шаблоны. (Да, это очередной пример, показывающий, как C++ подталкивает программиста к определению технических терминов, которые легко перепутать. Это еще одно маленькое дополнение к уже известным нам парам — оператор new и operator new, ковариантность и контрвариантность, const_iterator и константный итератор, — гарантирующим, что каждый обзор конструкции станет захватывающим приключением.) В самых лучших традициях тавтологии член-шаблон — это член, который представляет собой шаблон:

```
template <typename T>
class SList {
   public:
      //...
      template <typename In> SList( In begin, In end );
      //...
};
```

Этот конструктор SList, в отличие от конструктора по умолчанию, является членом-шаблоном, где имя типа In — явный параметр. Также он имеет неявный параметр — имя типа, используемое для создания экземпляра шаблона SList, членом которого он является. Этим объясняются повторения в описании члена-шаблона при его описании вне шаблона класса:

```
template <typename T> // для SList
template <typename In>    // для члена
SList<T>::SList( In begin, In end ) : head_( 0 ) {
```

```
      while( begin != end )
         push_front( *begin++ );
      reverse();
   }
```

Как и для других шаблонов функций, компилятор проведет логический вывод аргументов и создаст экземпляр шаблона конструктора, когда это будет необходимо (см. тему 57 «*Логический вывод аргументов шаблона*»):

```
float rds[] = { ... };
const int size = sizeof(rds)/sizeof(rds[0]);
std::vector<double> rds2( rds, rds+size );
//...
SList<float> data( rds, rds+size );              // In является float *
SList<double> data2( rds2.begin(), rds2.end() ); // In является
                                                 // vector<double>::iterator
```

Это обычное применение шаблонов конструкторов в STL, позволяющее инициализировать контейнер последовательностью значений, извлекаемых из произвольного источника. Члены-шаблоны также служат для генерирования функций-членов, подобных конструктору копий и оператору копирующего присваивания:

```
template <typename T>
class SList {
   public:
      //...
      template <typename S>
         SList( const SList<S> &that );
      template <typename S>

         SList &operator =( const SList<S> &rhs );
      //...
};
```

Эти члены шаблона могут применяться для операций, подобных конструктору копий и копирующему присваиванию:

```
SList<double> data3( data ); // T является double, S является float
data = data3;                // T является float, S является double
```

Обратите внимание на пространные определения «подобный конструктору копий» и «подобный копирующему присваиванию» в описании выше. Дело в том, что в приведенном выше примере присваивания не будет создан экземпляр члена-шаблона. То есть если приведенные выше `T` и `S` одного типа, компилятор не будет создавать член-шаблон, а «напишет» операцию копирования самостоятельно. В подобных случаях лучше всего явно

описывать операции копирования, чтобы предотвратить любезную и обычно неуместную помощь компилятора:

```
template <typename T>
class SList {
   public:
      //...
      SList( const SList &that );            // конструктор копий
      SList &operator =( const SList &rhs );  // копирующее присваивание
      template <typename S> SList( const SList<S> &that );
      template <typename S>
         SList &operator =( const SList<S> &rhs );
      //...
};
//...
SList<float> data4( data ); // конструктор копий
data3 = data2;              // копирующее присваивание
data3 = data4;              // некопирующее присваивание из шаблона члена
```

Любая невиртуальная функция-член может быть шаблоном (члены-шаблоны не могут быть виртуальными, потому что сочетание этих характеристик приводит к непреодолимым техническим проблемам при их реализации). Например, для нашего списка можно определить операцию сортировки:

```
template <typename T>
class SList {
   public:
      //...
      template <typename Comp> void sort( Comp comp );
      //...
};
```

Член-шаблон `sort` обеспечивает его пользователям возможность передавать указатель на функцию или объект-функцию, который будет использоваться для сравнения элементов списка (см. тему 20 «*Объекты-функции STL*»).

```
data.sort( std::less<float>() );     // восходящая сортировка
data.sort( std::greater<float>() );  // нисходящая сортировка
```

Здесь созданы экземпляры двух разных версий члена `sort` с помощью стандартных объектов-функций `less` (меньше) и `greater` (больше).

Тема 51 | Устранение неоднозначности с помощью ключевого слова template

В теме 49 «*Устранение неоднозначности с помощью имени типа*» было показано, как важно иногда для обеспечения правильного синтаксического разбора явно сообщить компилятору о том, что вложенное имя является именем типа. Аналогичная ситуация возникает со вложенными именами шаблонов.

Канонический пример – реализация распределителя STL. Если вы не знакомы с распределителями STL, ничего страшного. Пока это не обязательно, но, вероятно, потребуется много терпения.

Распределитель – это класс, используемый при настройке операций управления памятью для контейнеров STL. Распределители обычно реализуются как шаблоны классов:

```
template <class T>
class AnAlloc {
   public:
      //...
      template <class Other>
      class rebind {
         public:
            typedef AnAlloc<Other> other;
      };

      //...
};
```

Шаблон класса AnAlloc содержит вложенное имя rebind (повторное связывание), которое само по себе является шаблоном класса. В инфраструктуре STL он предназначен для создания распределителей, «аналогичных» распределителю, служащему для создания экземпляра контейнера, но для другого типа элемента. Например:

```
typedef AnAlloc<int> AI;                  // исходный распределитель
                                          // распределяет элементы типа int
typedef AI::rebind<double>::other AD;   // распределяет элементы типа double
typedef AnAlloc<double> AD;               // допустимо! тип тот же
```

Это может показаться странным, но механизм повторного связывания позволяет создавать версию существующего распределителя для разных типов элементов, не зная о типе распределителя или типе элемента.

```
typedef SomeAlloc::rebind<Node>::other NodeAlloc;
```

Если имя типа SomeAlloc соблюдает соглашение STL для распределителей, в нем должен быть вложенный шаблон класса rebind. Буквально это означает: «Я не знаю, каким распределителем является этот тип, и я не знаю, что он размещает, но хочу точно такой же распределитель для размещения элементов типа Node!».

Этот уровень неведения может иметь место только в рамках шаблона, где точные типы и значения неизвестны до момента создания экземпляра шаблона. Расширим контейнер SList из темы 50 «*Члены-шаблоны*», включив в него тип распределителя (A), который может размещать элементы (типа T). Как и все стандартные контейнеры, SList предоставит распределитель по умолчанию:

```
template < typename T, class A = std::allocator<T> >
class SList {
   //...
   struct Node {
      //...
   };
   typedef A::rebind<Node>::other NodeAlloc; // синтаксическася ошибка!
   //...
};
```

Как свойственно спискам и другим основанным на узлах контейнерам, наш список элементов типа T на самом деле не размещает элементы T и не управляет ими. Он работает с узлами, содержащими члены типа T. Данная ситуация описывалась выше. Имеется некоторый распределитель, знающий, как распределять объекты типа T, но требуется распределять объекты типа Node. Однако при попытке обращения к rebind возникает синтаксическая ошибка.

Повторюсь, проблема состоит в том, что компилятор не располагает в данной точке никакой информацией об имени типа A, кроме того, что это имя типа. Компилятор вынужден делать предположение о том, что вложенное имя rebind является именем нешаблона, и угловая скобка, следующая за ним, интерпретируется как знак «меньше чем». Но проблемы только начинаются. Даже если бы компилятор, дойдя до двукратно вложенного имени other, каким-то образом мог установить, что rebind — это имя шаблона, он был бы вынужден признать, что это не имя типа! Пора кое-что прояснить. typedef должен записываться так:

```
typedef typename A::template rebind<Node>::other NodeAlloc;
```

Ключевое слово template дает понять компилятору, что rebind — это имя шаблона, а typename сообщает компилятору, что вся эта ерунда является именем типа. Просто, правда?

Тема 52 | Создание специализации для получения информации о типе

Явная и частичная специализации шаблона класса обычно служат для создания версий базового шаблона класса, настроенных под конкретные аргументы или классы аргументов шаблона (см. темы 46 «*Явная специализация шаблона класса*» и 47 «*Частичная специализация шаблонов*»).

Однако эти возможности языка также широко применяются для обратных операций, когда происходит не создание специализации на основании свойств типа, а свойства типа логически выводятся из специализации. Рассмотрим простой пример:

```
template <typename T>
struct IsInt                    // T не типа int...
   { enum { result = false }; };
template <>
struct IsInt<int>    // если только это int!
   { enum { result = true }; };
```

Прежде чем продолжить, хотелось бы обратить внимание на то, насколько прост приведенный выше код (если вы разобрались с его витиеватым синтаксисом). Это простой пример метапрограммирования шаблонов, т. е. осуществления некоторой части вычислений во время компиляции, а не во время выполнения, посредством создания экземпляра шаблона. Звучит довольно заумно, но сводится к наблюдению, которое могли бы сделать даже мои простодушные соседи, занимающиеся выращиванием клюквы: «Это `int`, если это `int`». Самые сложные аспекты программирования на C++ с применением шаблонов не сложнее этого, просто они более запутанные.

Имея базовый шаблон и полную специализацию, можно спросить (во время компиляции), не является ли неизвестный тип типом int:

```
template <typename X>
void aFunc( X &arg ) {
   //...
   ...IsInt<X>::result...
   //...
}
```

Возможность задавать подобные вопросы о типах во время компиляции – основа для ряда важных методов оптимизации и контроля ошибок. Конечно, информация о том, что какой-то тип точно int, на практике не очень полезна. А вот знать, что тип является указателем, вероятно, более выгодно, поскольку реализации, использующие указатели на объекты или непосредственно объекты, отличаются друг от друга:

```
struct Yes {};             // тип, аналогичный true
struct No {};              // тип, аналогичный false

template <typename T>
struct IsPtr               // T не является указателем...
   { enum { result = false }; typedef No Result; };
template <typename T>
struct IsPtr<T *>          // если только это неквалифицированный указатель,
   { enum { result = true }; typedef Yes Result; };
template <typename T>
struct IsPtr<T *const>     // или константный указатель,
   { enum { result = true }; typedef Yes Result; };
template <typename T>
struct IsPtr<T *volatile>  // или изменяемый указатель,
   { enum { result = true }; typedef Yes Result; };
template <typename T>
struct IsPtr<T *const volatile>  // или константный изменяемый указатель.
   { enum { result = true }; typedef Yes Result; };
```

В случае применения IsPtr задается более общий вопрос, чем для IsInt. Таким образом, частичная специализация применяется для «выявления» по-разному квалифицированных вариантов модификатора указателя. Как сказано выше, эта возможность IsPtr на самом деле не намного сложнее для понимания, чем IsInt. Просто ее синтаксис более запутанный. (Подобная техника метапрограммирования также приведена в теме 59 «*Концепция SFINAE*».)

Чтобы увидеть практическую выгоду от возможности задавать вопросы о типе во время компиляции, рассмотрим данную реализацию простого шаблона стека:

```
template <typename T>
class Stack {
   public:
      ~Stack();
      void push( const T &val );
      T &top();
      void pop();
      bool empty() const;
   private:
      //...
      typedef std::deque<T> C;
      typedef typename C::iterator I;
      C s_;
};
```

Наш стек – это симпатичный интерфейс-обертка для стандартной очереди с двумя концами deque. Аналогичный результат можно было бы получить, обратившись к стандартному адаптеру контейнера stack. Большинство операций очень просты и могут быть реализованы непосредственно с помощью deque.

```
template <typename T>
void Stack<T>::push( const T &val )
   { s_.push_back( val ); }
```

Но могут возникнуть проблемы с деструктором Stack. При уничтожении Stack уничтожается и его элемент данных deque, что, в свою очередь, приведет к уничтожению всех элементов, остающихся в deque. Однако если эти элементы представляют собой указатели, то объекты, на которые они указывают, удалены не будут. Таково поведение стандартного контейнера deque. Поэтому необходимо выбрать политику уничтожения элементов указателей для Stack, который будет приговорен к уничтожению! (Более гибкий подход представлен в теме 56 «*Политики*».) Мы не можем поручить уничтожение элементов deque деструктору, потому что это приведет к ошибке в тех случаях, когда элементы не являются указателями.

Одним из возможных решений могла бы быть частичная специализация (базового) шаблона Stack для обработки стеков указателей (см. тему 47 «*Частичная специализация шаблонов*»). Однако это, наверное, чересчур, если требуется лишь немного изменить поведение Stack. Совсем другое дело просто задавать очевидный вопрос (во время компиляции) и действовать соответственно ответу: «Если тип элементов, хранящихся в Stack, представляет собой указатель, то уничтожить все остающиеся элементы. В противном случае не уничтожать».

```
template <typename T>
class Stack {
```

```
    public:
        ~Stack()
            { cleanup( typename IsPtr<T>::Result() ); }
        //...
    private:
        void cleanup( Yes ) {
            for( I i( s_.begin() ); i != s_.end(); ++i )
                delete *i;
        }
        void cleanup( No )
            {}
        typedef std::deque<T> C;
        typedef typename C::iterator I;
        C s_;
};
```

Здесь мы имеем две разные функции-члена cleanup («очистка»), одна из которых принимает аргумент типа Yes (Да), тогда как другая принимает аргумент типа No (Нет). Версия Yes осуществляет уничтожение; версия No – нет. Деструктор задает вопрос: «Является ли T указателем?». Фактически для этого создается экземпляр IsPtr с T, организуется доступ к вложенному имени типа Result (Результат) (см. тему 49 «*Устранение неоднозначности с помощью ключевого слова typename*»), которым может быть Yes или No, и объект этого типа передается в cleanup. Будет создан и вызван экземпляр только одной из двух версий cleanup (см. тему 61 «*Мы создаем экземпляр того, что используем*»).

```
Stack<Shape *> shapes;    // будет уничтожать
Stack<std::string> names; // не будет уничтожать
```

Специализации шаблонов классов могут служить для извлечения из типов информации любого уровня сложности. Например, может понадобиться знать не только о том, является ли определенный тип массивом, но и, если это массив, какие элементы в нем хранятся и каков его размер:

```
template <typename T>
struct IsArray {          // T не является массивом...
    enum { result = false };
    typedef No Result;
};
template <typename E, int b>
struct IsArray<E [b]> {  // ...если только он массив!
    enum { result = true };
    typedef Yes Result;
    enum { bound = b };   // граница массива
    typedef E Etype;      // тип элемента массива
};
```

Может понадобиться информация не только о том, является ли определенный тип указателем на данные-член, но и, если это так, имена типов класса и члена, на которые он указывает:

```
template <typename T>
struct IsPCM {              // T не является указателем на данные-член
   enum { result = false };
   typedef No Result;
};
template <class C, typename T>
struct IsPCM<T C::*> {    // ...если он является!
   enum { result = true };
   typedef Yes Result;
   typedef C ClassType;   // класс
   typedef T MemberType;  // тип члена класса
};
```

Эти методики применяются во многих популярных инструментальных средствах, предоставляющих возможность доступа к свойствам типа (см. тему 54 «*Свойства*») для настройки кода во время компиляции.

Тема 53 | Встроенная информация о типе

Как узнать тип элементов контейнера?

```
template <typename T>
class Seq {
   //...
};
```

На первый взгляд это совсем нетрудно. Элемент `Seq<std::string>` относится к типу `std::string`, правильно? Необязательно. При реализации (нестандартного) контейнера последовательности ему ничто не мешает иметь тип `const T`, `T *` или «умный указатель на `T`». (Какой-нибудь особенно замысловатый контейнер мог бы просто игнорировать параметр шаблона и всегда задавать для элемента тип `void *`!) Но капризы реализации не единственная причина, мешающая определить тип элемента контейнера. Часто создается универсальный код, в котором эта информация просто недоступна.

```
template <class Container>
Elem process( Container &c, int size ) {
   Temp temp = Elem();
   for( int i = 0; i < size; ++i )
      temp += c[i];
   return temp;
}
```

В приведенном выше универсальном алгоритме `process` требуется знать тип элемента (`Elem`) контейнера (`Container`), а также тип, который мог бы служить для объявления временного хранилища объектов типа элемента (`Temp`). Но эта информация недоступна, пока не создан экземпляр шаблона функции `process` с определенным контейнером.

Обычный выход из данной ситуации – заставить тип предоставить «личную» информацию о себе. Такая информация обычно встраивается в сам тип. Это напоминает внедрение микрочипа в человека, с которого можно считать данные об его имени, идентификационном номере, группе крови и т. д. (Это лишь аналогия, а не оправдание подобных вещей.) Нас не интересует группа крови последовательности, но необходимо знать тип ее элементов.

```
template <class T>
class Seq {
   public:
      typedef T Elem;  // тип элемента
      typedef T Temp;  // тип временного хранилища
      size_t size() const;
      //...
};
```

Эту встроенную информацию можно запросить во время компиляции:

```
typedef Seq<std::string> Strings;
//...
Strings::Elem aString;
```

Такой подход знаком любому пользователю стандартной библиотеки контейнеров. Например, чтобы объявить итератор для стандартного контейнера, рекомендуется узнать у самого контейнера тип его итератора:

```
vector<int> aVec;
//...
for( vector<int>::iterator i( aVec.begin() );
    i != aVec.end(); ++i )
    //...
```

Здесь мы не предполагаем, что типом итератора является `int *` (как это часто бывает во многих реализациях), а просим `vector<int>` сообщить тип своего итератора. Типом итератора для `vector<int>` может быть любой другой тип (например, пользовательский безопасный тип указателя). Таким образом, единственный способ написать переносимый цикл – получить тип итератора от самого `vector<int>`.

Более важное наблюдение – данный подход позволяет писать универсальный код, делающий *предположение* о наличии требуемой информации.

```
template <class Container>
typename Container::Elem process( Container &c, int size ) {
   typename Container::Temp temp
       = typename Container::Elem();
   for( int i = 0; i < size; ++i )
       temp += c[i];
```

```
    return temp;
}
```

Эта версия алгоритма process запрашивает у типа Container его личные данные и предполагает, что Container определяет вложенные имена типов Elem и Temp. (Три ключевых слова typename явно указывают компилятору на то, что вложенные имена — это имена типов, а не какие-то другие вложенные имена. См. тему 49 «*Устранение неоднозначности с помощью имени типа*».)

```
Strings strings;
aString = process( strings, strings.size() ); // OK
```

Алгоритм process хорошо работает с контейнером Seq и будет работать с любым другим контейнером, соблюдающим наше соглашение.

```
template <typename T>
class ReadonlySeq {
   public:
      typedef const T Elem;
      typedef T Temp;
      //...
};
```

Мы можем применить process к контейнеру ReadonlySeq, потому что он удовлетворяет нашим предположениям.

Тема 54 | Свойства

Иногда недостаточно знать тип объекта. Часто существует связанная с типом объекта информация, исключительно важная для работы с объектом. В теме 53 «*Встроенная информация о типе*» было показано, что такие сложные типы, как стандартные контейнеры, часто содержат встроенную информацию о себе:

```
Strings strings;
aString = process( strings, strings.size() );   // OK
std::vector<std::string> strings2;
aString = process( strings2, strings2.size() ); // ошибка!
extern double readings[RSIZ];
double r = process( readings, RSIZ );           // ошибка!
```

Алгоритм `process` эффективен в случае контейнера `Seq`, но не годится для стандартного контейнера `vector`, потому что `vector` не определяет вложенные имена типов, существование которых предполагает `process`.

Алгоритм `process` может применяться к контейнеру `ReadonlySeq`, потому что он удовлетворяет нашим предположениям. Но может понадобиться применить `process` к контейнерам, не следующим этому довольно узкому соглашению, или к «контейнероподобным» объектам, даже не являющимся классами. Часто для решения этих проблем привлекаются *классы свойств* (*traits classes*).

Класс свойств – это коллекция информации о типе. Однако в отличие от вложенной информации контейнера, класс свойств не зависит от типа, который описывает.

```
template <typename Cont>
struct ContainerTraits;
```

Классы свойств широко применяются для размещения отвечающего соглашению слоя между универсальными алгоритмами и типами, которые не следуют соглашению. Алгоритм создан с учетом особенностей типа. Общий случай всегда предполагает наличие какого-то соглашения. В данном случае ContainerTraits принимает соглашение, используемое контейнерами Seq и ReadonlySeq.

```
template <typename Cont>
struct ContainerTraits {
   typedef typename Cont::Elem Elem;
   typedef typename Cont::Temp Temp;
   typedef typename Cont::Ptr Ptr;
};
```

С введением этого шаблона класса свойств появляется выбор. Ссылаться на вложенный тип Elem одного из наших типов контейнеров теперь можно или через тип контейнера, или через экземпляр типа свойств, созданный для типа контейнера.

```
typedef Seq<int> Cont;
Cont::Elem e1;
ContainerTraits<Cont>::Elem e2;  // того же типа, что и e1
```

Наш универсальный алгоритм можно переписать с использованием свойств вместо прямого доступа к именам вложенных типов контейнера:

```
template <typename Container>
typename ContainerTraits<Container>::Elem
process( Container &c, int size ) {
   typename ContainerTraits<Container>::Temp temp
      = typename ContainerTraits<Container>::Elem();
   for( int i = 0; i < size; ++i )
      temp += c[i]; return temp;
}
```

Может показаться, что мы лишь усложнили синтаксис универсального алгоритма process! Ранее, чтобы получить тип элемента контейнера, мы записывали typename Container::Elem. В переводе на обычный язык это означает: «Получи вложенное имя Elem контейнера Container. Кстати, это имя типа». С классами свойств приходится записывать typename ContainerTraits<Container>::Elem. Это означает: «Создай экземпляр класса ContainerTraits, соответствующего этому контейнеру, и получи его вложенное имя Elem. Кстати, это имя типа». Сделан шаг назад – мы получаем информацию не от типа контейнера непосредственно, а от посредника, в виде класса свойств. Если доступ к вложенной информации типа рассматривать как считывание информации о человеке со встроенного микрочипа, то использование класса свойств аналогично поиску информации о ком-то в базе

данных с применением имени человека в качестве ключа. В результате получаем одну и ту же информацию, но поиск в базе данных, конечно, менее агрессивный и более гибкий.

Например, нельзя получить информацию с микрочипа, если его нет. (Возможно, человек прибыл из региона, где встроенные микрочипы носить не обязательно.) Однако для такого человека всегда можно создать новую запись в базе данных, даже не проинформировав его об этом. Точно так же можно специализировать шаблон свойств, чтобы он предоставлял информацию о конкретном не соответствующем соглашениям контейнере, не касаясь самого контейнера:

```
class ForeignContainer {
   // нет вложенной информации типа...
};
//...
template <>
struct ContainerTraits<ForeignContainer> {
   typedef int Elem;
   typedef Elem Temp;
   typedef Elem *Ptr;
};
```

Имея в распоряжении специализацию ContainerTraits, можно применять алгоритм process к ForeignContainer так же эффективно, как к контейнеру, написанному в соответствии с нашим соглашением. Первоначальная реализация process в применении к ForeignContainer даст сбой, потому что попытается получить доступ к несуществующей вложенной информации:

```
ForeignContainer::Elem x;         // ошибка, нет такого вложенного имени!
```

```
ContainerTraits<ForeignContainer>::Elem y;  // OK, использование свойств
```

Полезно рассматривать шаблон свойств как коллекцию информации, индексированную типом, по аналогии с ассоциативным контейнером, индексированным ключом. Но «индексация» свойств происходит во время компиляции через специализацию шаблона.

Другое преимущество доступа к информации о типе через класс свойств в том, что эта методика может применяться для предоставления информации о типах, не являющихся классами и, следовательно, не имеющих вложенной информации. Даже несмотря на то, что классы свойств являются классами, типы, свойства которых они инкапсулируют, не обязаны быть таковыми. Например, массив — это своего рода (математически или фактически) вырожденный контейнер, с которым можно работать как с обычным контейнером.

```
template <>
struct ContainerTraits<const char *> {
```

```
    typedef const char Elem;
    typedef char Temp;
    typedef const char *Ptr;
};
```

Имея в распоряжении эту специализацию для «контейнера» типа const char *, можно применять process к массиву символов.

```
const char *name = "Arsene Lupin";
const char *r = process( name, strlen(name) );
```

Можно продолжать в том же духе для других типов массивов, создавая специализации для int *, const double * и т. д. Однако было бы удобнее определить один вариант для любого типа указателей, поскольку все они будут иметь общие свойства. Этой цели служит частичная специализация шаблона свойств для указателей:

```
template <typename T>
struct ContainerTraits<T *> {
    typedef T Elem;
    typedef T Temp;
    typedef T *Ptr;
};
```

Специализация ContainerTraits для любого типа указателя, будь то int * или const float *(*const *)(int), приведет к созданию экземпляра этой частичной специализации, если только не доступна еще более специализированная версия ContainerTraits.

```
extern double readings[RSIZ];
double r = process( readings, RSIZ ); // работает!
```

Однако это еще не все. Обратите внимание, что в случае частичной специализации для указателя на константу будет неправильно выбран «временный» тип. То есть константные временные значения не имеют особого практического значения, потому что для них нельзя осуществить операцию присваивания. Желательно было бы иметь в качестве временного типа неконстантный аналог типа элемента. В случае с const char *, например, ContainerTraits<const char *>::Temp должен иметь тип char, а не const char. Выйти из этой ситуации можно с помощью частичной специализации.

```
template <typename T>
struct ContainerTraits<const T *> {
    typedef const T Elem;
    typedef T Temp;              // примечание: неконстантный аналог Elem
    typedef const T *Ptr;
};
```

Эта более специализированная частичная специализация будет предпочтительнее предыдущей в тех случаях, когда аргумент шаблона представляет собой указатель на константу, а не указатель на неконстанту.

Частичная специализация также способна помочь в расширении механизма свойств для приведения «чужого» соглашения в соответствие с локальным соглашением. Например, STL очень жестко придерживается соглашений (см. тему 4 «*Стандартная библиотека шаблонов*»), и стандартные контейнеры имеют понятия, подобные тем, которые инкапсулирует наш ContainerTraits, но они иначе представлены. Вернемся к неудачной попытке создания экземпляра алгоритма process для стандартного vector и исправим ошибки.

```
template <class T>
struct ContainerTraits< std::vector<T> > {
   typedef typename std::vector<T>::value_type Elem;
   typedef typename
      std::iterator_traits<typename
      std::vector<T>::iterator>
      ::value_type Temp;
   typedef typename
      std::iterator_traits<typename
      std::vector<T>::iterator>
      ::pointer Ptr;
};
```

Это не самая удобочитаемая реализация из тех, которые можно представить, но она скрытая, и теперь пользователи могут инициировать наш универсальный алгоритм с помощью контейнера, созданного из стандартного vector.

```
std::vector<std::string> strings2;
aString = process( strings2, strings2.size() );  // работает!
```

Тема 55 | Параметры-шаблоны шаблона

Вернемся к шаблону Stack, рассматриваемому в теме 52 «*Создание специализации для получения информации о типе*». Он был реализован с применением стандартного контейнера deque. Это довольно хороший компромисс для реализации, хотя во многих случаях более эффективным или подходящим мог бы быть другой контейнер. Данную проблему можно решить введением в Stack дополнительного параметра шаблона для типа контейнера, задействованного в его реализации.

```
template <typename T, class Cont>
class Stack;
```

Для простоты откажемся от стандартной библиотеки (далеко не всегда это правильный путь) и предположим, что имеем в своем распоряжении ряд нестандартных шаблонов контейнеров: List, Vector, Deque и, возможно, другие. Представим, что эти контейнеры аналогичны стандартным, но имеют только один параметр шаблона для типа элемента контейнера.

Вспомним, что на самом деле у стандартных контейнеров как минимум два параметра: тип элемента и тип распределителя. Распределители необходимы контейнерам для того, чтобы иметь возможность настраивать процессы выделения и высвобождения своей рабочей памяти. Иначе говоря, распределитель определяет политику управления памятью для контейнера (см. тему 56 «*Политики*»). Распределитель имеет значение по умолчанию, так что о нем легко забыть. Однако при создании экземпляра стандартного контейнера, такого как vector<int>, фактически получаем vector< int, std::allocator<int> >.

Например, объявление нестандартного List было бы таким:

```
template <typename> class List;
```

Обратите внимание, что в приведенном выше объявлении List пропущено имя параметра шаблона. То же самое происходит с именем формального аргумента в объявлении функции: задавать имя параметра шаблона в объявлении шаблона не обязательно. Аналогично с описанием функции: имя параметра шаблона требуется только в описании шаблона и только если имя параметра фигурирует в шаблоне. Однако, как и для формальных аргументов в объявлениях функций, в объявлениях шаблонов параметрам шаблонов обычно даются имена, чтобы облегчить документирование шаблона.

```
template <typename T, class Cont>
class Stack {
   public:
      ~Stack();
      void push( const T & );
      //...
   private:
      Cont s_;
};
```

Теперь пользователь Stack должен предоставить два аргумента шаблона – тип элемента и тип контейнера, а контейнер должен быть в состоянии хранить объекты типа элемента.

```
Stack<int, List<int> > aStack1;             // OK
Stack<double, List<int> > aStack2;          // допустимо, но не годится
Stack<std::string, Deque<char *> > aStack3;  // ошибка!
```

В случае aStack2 и aStack3 возникают проблемы с согласованностью. Если пользователь выбирает неверный тип контейнера для типа элемента, возникает ошибка компиляции (в случае aStack3 из-за невозможности копировать string в char *) или скрытый дефект (в случае aStack2 из-за погрешности при копировании double в int). Кроме того, большинство пользователей Stack не хотят возиться с выбором базовой реализации и будут удовлетворены разумным значением по умолчанию. Ситуацию можно улучшить, предоставив значение по умолчанию для второго параметра шаблона:

```
template <typename T, class Cont = Deque<T> >
class Stack {
   //...
};
```

Это помогает, когда пользователь Stack желает принять реализацию Deque или реализация его не особо волнует.

```
Stack<int> aStack1;       // контейнер - Deque<int>
Stack<double> aStack2;    // контейнер - Deque<double>
```

Этот подход приблизительно похож на тот, который используется адаптерами стандартных контейнеров stack, queue и priority_queue.

```
std::stack<int> stds;  // контейнер - deque< int, allocator<int> >
```

Здесь мы видим разумный компромисс между удобством случайного использования возможностей Stack и гибкостью для опытного пользователя, которая обеспечивает возможность использования любого (допустимого и эффективного) контейнера для хранения элементов Stack.

Однако эта гибкость достигается за счет надежности. По-прежнему необходимо согласовывать типы элемента и контейнера в других специализациях, и это требование согласования открывает простор для ошибок.

```
Stack<int, List<int> > aStack3;
Stack<int, List<unsigned> > aStack4; // ой!
```

Давайте посмотрим, можно ли увеличить надежность и сохранить при этом приемлемую гибкость. Шаблон может принимать параметр, сам являющийся именем шаблона. Эти параметры имеют замечательное название — ***параметры-шаблоны шаблона*** (***template template parameters***).

```
template <typename T, template <typename> class Cont>
class Stack;
```

При виде этого нового списка параметров шаблона для класса Stack просто опускаются руки, но все не так плохо, как кажется. Первый параметр, T, был здесь и раньше. Это просто имя типа. Второй параметр, Cont — параметр-шаблон шаблона. Это имя шаблона класса с одним параметром — именем типа. Заметьте, что для параметра имени типа Cont имя не задано, хотя это можно было бы сделать:

```
template <typename T, template <typename ElementType> class Cont>
class Stack;
```

Однако такое имя (ElementType (Тип элемента)) может играть только информативную роль как имя формального аргумента в объявлении функции. Обычно эти имена опускаются, но могут свободно использоваться везде, где улучшают читаемость кода. Наоборот, можно было бы воспользоваться возможностью и снизить читаемость до минимума, исключив из объявления Stack все необязательные с технической точки зрения имена:

```
template <typename, template <typename> class>
class Stack;
```

Но из сочувствия к тем, кому придется читать код, надо избегать такой практики, даже несмотря на то, что язык C++ позволяет делать это.

Шаблон Stack использует свой параметр имени типа для создания экземпляра параметра-шаблона. Получающийся в результате тип контейнера служит для реализации Stack:

```
template <typename T, template <typename> class Cont> class Stack {
      //...
   private:
      Cont<T> s_;
};
```

При таком подходе согласование элемента и контейнера может осуществляться самой реализацией Stack, а не кодом, специализирующим Stack. Но этот момент специализации сокращает возможность ошибки при согласовании типа элемента и контейнера, служащего для хранения элементов.

```
Stack<int,List> aStack1;
Stack<std::string,Deque> aStack2;
```

Для дополнительного удобства можно использовать значение по умолчанию аргумента-шаблона шаблона:

```
template <typename T, template <typename> class Cont = Deque>
class Stack {
   //...
};
//...
Stack<int> aStack1;               // используется значение по умолчанию:
                                  // Cont является Deque
Stack<std::string,List> aStack2; // Cont является List
```

Обычно этот подход хорош для согласования набора аргументов шаблона и шаблона, экземпляр которого должен быть создан с участием этих аргументов.

Часто параметры-шаблоны путают с параметрами имени типа, которые по чистой случайности генерируются из шаблонов. Рассмотрим следующее объявление шаблона класса:

```
template <class Cont> class Wrapper1;
```

Шаблон Wrapper1 (Обертка1) принимает имя типа в качестве аргумента шаблона. (В объявлении параметра Cont шаблона Wrapper1 вместо ключевого слова typename используется ключевое слово class, чтобы сообщить читателям о том, что здесь ожидается class или struct, а не произвольный тип. Хотя компилятору все равно. В данном контексте технически typename и class означают абсолютно одно и то же. См. тему 63 «*Необязательные ключевые слова*».) Это имя типа могло бы быть сгенерировано из шаблона, как в Wrapper1<List<int>>, но List<int> по-прежнему остается

всего лишь именем класса, даже несмотря на то, что был сгенерирован из шаблона.

```
Wrapper1< List<int> > w1;      // хорошо, List<int> - имя типа
Wrapper1< std::list<int> > w2;  // хорошо, list<int> - тип
Wrapper1<List> w3;              // ошибка! List – имя шаблона
```

В качестве альтернативы рассмотрим следующее объявление шаблона класса:

```
template <template <typename> class Cont> class Wrapper2;
```

Шаблону Wrapper2 требуется имя шаблона как аргумент шаблона, и не просто имя шаблона. Объявление гласит, что шаблон должен принимать один аргумент имени типа.

```
Wrapper2<List> w4;             // хорошо, List – шаблон с одним аргументом
Wrapper2< List<int> > w5;      // ошибка! List<int> не шаблон
Wrapper2<std::list> w6;        // ошибка! std::list принимает два
                               // и более аргументов
```

Если мы хотим иметь шанс создавать специализации со стандартным контейнером, необходимо сделать следующее:

```
template <template <typename Element,
             class Allocator> class Cont>
class Wrapper3;
```

или, что равнозначно:

```
template <template <typename,typename> class Cont>
class Wrapper3;
```

Это объявление говорит о том, что шаблон должен принимать два аргумента имени типа:

```
Wrapper3<std::list> w7;         // может работать...
Wrapper3< std::list<int> > w8;  // ошибка! list<int> - это класс
Wrapper3<List> w9;             // ошибка! List принимает один аргумент имени типа
```

Однако для шаблонов стандартных контейнеров (таких, как этот) допускается объявление более двух параметров, так что приведенное выше объявление w7 на некоторых платформах может не работать. Да, мы все любим и уважаем STL, но никогда не утверждали, что она идеальна.

Тема 56 | Политики

В теме 52 «*Создание специализации для получения информации о типе*» был спроектирован шаблон стека, который удалял все элементы, остающиеся в нем в конце существования, если типом элементов стека был указатель.

```
template <typename T> class Stack;
```

Такая политика не лишена здравого смысла, но ей не хватает гибкости. Возможны ситуации, когда пользователь не хочет удалять то, на что ссылаются указатели стека. Например, указатели могут ссылаться на объекты, не находящиеся в куче или используемые совместно с другими контейнерами. Кроме того, указатель может ссылаться на массив объектов, а не на одиночный объект. Если имеется стек указателей на символы, практически со стопроцентной уверенностью можно сказать, что так оно и есть, потому что указатели символов обычно ссылаются на NTCTS (null terminated array of characters – массивы символов, завершающиеся символом null):

```
Stack<const char *> names;    // ой! неопределенное поведение
```

Наша политика уничтожения предполагает, что указатели Stack ссылаются на одиночный объект, и, следовательно, использует форму оператора delete, предназначенную для удаления одиночных объектов, тогда как для массива должна применяться специальная форма оператора delete: operator delete[] (см. тему 37 «*Создание массивов*»).

Наша цель – суметь написать примерно такой деструктор шаблона Stack:

```
template <typename T>
class Stack {
   public:
```

```
        ~Stack() {
            for( I i( s_.begin() ); i != s_.end(); ++i )
                doDeletionPolicy( *i );
        }
        //...
    private:
        typedef std::deque<T> C;
        typedef typename C::iterator I;
        C s_;
};
```

Деструктор перебирает все оставшиеся элементы и применяет предусмотренную для каждого элемента политику уничтожения. Функцию `doDeletionPolicy` (примени политику уничтожения) можно реализовать по-разному. Обычно политика задается явно при создании экземпляра шаблона `Stack` и реализуется параметром-шаблоном шаблона (см. тему 55 «*Параметры-шаблоны шаблона*»).

```
template <typename T, template <typename> class DeletionPolicy>
class Stack {
    public:
        ~Stack() {
            for( I i( s_.begin() ); i != s_.end(); ++i )
                DeletionPolicy<T>::doDelete( *i );  // выполняется политика
        }
        //...
    private:
        typedef std::deque<T> C;
        typedef typename C::iterator I;
        C s_;
};
```

Рассмотрев применение политики уничтожения в деструкторе `Stack`, можно установить, что политика уничтожения `Stack` – это шаблон класса, экземпляр которого создается вместе с типом элемента `Stack`. Имеется статическая функция-член `doDelete`, осуществляющая соответствующее действие по уничтожению элемента `Stack`. Теперь можно приступить к определению некоторых необходимых политик. Одна из них – политика уничтожения:

```
template <typename T>
struct PtrDeletePolicy {
    static void doDelete( T ptr )
        { delete ptr; }
};
```

Конечно, можно было бы реализовать политику с помощью другого интерфейса. Например, можно было бы обойтись без статической функции, а перегрузить оператор вызова функции:

```
template <typename T>
struct PtrDeletePolicy {
   void operator ()( T ptr )
       { delete ptr; }
};
```

и изменить операцию уничтожения в деструкторе Stack

```
DeletionPolicy<T>()(*i);
```

Важно установить соглашение, требующее, чтобы доступ к реализации каждой политики осуществлялся с использованием одного и того же синтаксиса.

Другие полезные политики выполняют уничтожение массивов или вообще ничего не делают:

```
template <typename T>
struct ArrayDeletePolicy {
   static void doDelete( T ptr )
      { delete [] ptr; }
};
template <typename T>
struct NoDeletePolicy {
   static void doDelete( const T & )
      {}
};
```

Теперь при создании экземпляра Stack можно задать соответствующую политику уничтожения:

```
Stack<int, NoDeletePolicy> s1;             // не удалять элементы типа int
Stack<std::string *, PtrDeletePolicy> s2;  // удалять элементы типа string *
Stack<const char *, ArrayDeletePolicy> s3; // использовать delete []
Stack<const char *, NoDeletePolicy> s4;    // не удалять!
Stack<int, PtrDeletePolicy> s5; // ошибка! не может удалять элемент типа int!
```

Если одна политика применяется чаще других, то ее и надо сделать политикой по умолчанию:

```
template <typename T,
   template <typename> class DeletionPolicy = NoDeletePolicy>
class Stack;
//...
Stack<int> s6;              // не уничтожать
Stack<const char *> s7;     // не уничтожать
Stack<const char *, ArrayDeletePolicy> s8; // delete []
```

Конструкция шаблона обычно предлагает несколько возможностей параметризации через политики. Например, в теме 55 «*Параметры-шаблоны шаблона*» пользователю была предоставлена возможность задавать способ реализации Stack. Это политика реализации:

```
template <typename T,
    template <typename> class DeletionPolicy = NoDeletePolicy
    template <typename> class Cont = Deque>
class Stack;
```

Политика реализации дает пользователю Stack дополнительную гибкость:

```
Stack<double *, ArrayDeletePolicy, Vector> dailyReadings;
```

обеспечивая при этом хорошее общее поведение по умолчанию.

```
Stack<double> moreReadings;   // нет уничтожения, используйте Deque
```

В универсальном проектировании обычно принимаются решения о политике реализации и поведения. Часто эти решения можно вынести и представить как политики.

Тема 57 | Логический вывод аргументов шаблона

Шаблоны классов должны специализироваться явно. Например, при специализации контейнера Heap, обсуждаемого в теме 46 «*Явная специализация шаблона класса*», необходимо предоставить шаблону аргумент, задающий тип:

```
Heap<int> aHeap;
Heap<const char *> anotherHeap;
```

Шаблоны функций также могут специализироваться явно. Предположим, имеется шаблон функции, осуществляющий ограниченное приведение в старом стиле:

```
template <typename R, typename E>
R cast( const E &expr ) {
   // ...проводится интеллектуальная проверка...
   return R( expr );    // ...и приведение.
}
```

Провести специализацию этого шаблона можно явно при его вызове, точно так же, как должен специализироваться шаблон класса:

```
int a = cast<int,double>(12.3);
```

Однако обычно удобнее позволить компилятору вывести аргументы шаблона из типов аргументов (фактических параметров) вызова функции. Ничего удивительного, этот процесс называется *логическим выводом аргументов шаблона*. Будьте осторожны! В приведенном ниже описании обратите внимание на разницу между терминами «аргумент шаблона» и «аргумент функции» (см. тему 45 «*Терминология шаблонов*»). Рассмот-

рим пример с одним аргументом шаблона, выявляющий меньший из двух аргументов функции.

```
template <typename T>
T min( const T &a, const T &b )
    { return a < b ? a : b; }
```

Когда `min` используется без явного предоставления аргументов шаблона, компилятор проверяет типы аргументов вызова функции, чтобы логически вывести аргумент шаблона:

```
int a = min( 12, 13 );          // T является int
double d = min( '\b', '\a' ); // T является char
char c = min( 12.3, 4 );        // ошибка! T не может быть одновременно
                                // и double, и int
```

Приведенная выше строка с ошибкой является результатом неспособности компилятора вывести аргумент шаблона в ситуации неоднозначности. В подобных случаях всегда есть возможность задать аргумент шаблона явно:

```
d = min<double>( 12.3, 4 );  // OK, T является double
```

Аналогичная ситуация возникает и с шаблоном `cast`, если попытаться использовать логический вывод аргумента шаблона:

```
int a = cast( 12.3 );         // ошибка! E - double, но чем является R?
```

Как и при разрешении перегрузки, при логическом выводе аргументов шаблона компилятор проверяет только типы аргументов функции, но не предполагаемые возвращаемые типы. Компилятор может узнать возвращаемый тип, только если мы сообщим его:

```
int a = cast<int>( 12.3 );  // E - double, R - int (задан явно)
```

Обратите внимание, что все прослеживаемые аргументы шаблона могут не включаться в список аргументов шаблона, если компилятор может логически вывести их. В данном случае компилятору был предоставлен только возвращаемый тип, а тип выражения он определил самостоятельно. Огромное значение для удобства в работе имеет порядок параметров шаблона, поскольку, если бы тип выражения предшествовал возвращаемому типу, пришлось бы задавать явно оба.

Здесь многие читатели обратят внимание на синтаксис приведенного выше вызова функции `cast` и спросят: «Вы подразумеваете, что `static_cast`, `dynamic_cast`, `const_cast` и `reinterpret_cast` являются шаблонами функций?». Нет, мы не делаем такого предположения, потому что эти четыре оператора приведения не шаблоны, а встроенные операторы (как опера-

торы new или оператор + для операций над целыми). Но, несомненно, выглядит так, как будто их синтаксис испытал влияние шаблона функции cast. (См. тему 9 «*Новые операторы приведения*».)

Обратите внимание, что логический вывод аргументов шаблона осуществляется путем проверки типов фактических параметров вызова. Тем самым подразумевается, что любой аргумент шаблона функции, который не может быть логически выведен из типов аргументов, должен задаваться явно. Например, вот шаблон функции, обеспечивающий надоедливые повторы:

```
template <int n, typename T>
void repeat( const T &msg ) {
   for( int i = 0; i < n; ++i )
       std::cout << msg << std::flush;
}
```

Здесь все сделано аккуратно и аргумент шаблона типа int помещен перед аргументом типа. Поэтому можно было обойтись заданием только числа повторов сообщения и позволить установить тип сообщения путем логического вывода аргументов шаблона:

```
repeat<12>( 42 );         // n - это 12, T - это int
repeat<MAXINT>( '\a' );   // n - это очень много, T - это char
```

В шаблонах cast, min, и repeat компилятор выводит единственный аргумент шаблона из единственного аргумента функции. Однако механизм логического вывода способен находить несколько аргументов шаблона из типа единственного аргумента функции:

```
template <int bound, typename T>
void zeroOut( T (&ary)[bound] ) {
   for( int i = 0; i < bound; ++i )
      ary[i] = T();
}
//...
const int hrsinweek = 7 * 24;
float readings[hrsinweek];
zeroOut( readings );     // bound == 168, T является float
```

В данном случае zeroOut ожидает в качестве аргумента массив, и механизм логического вывода аргументов может провести анализ типа аргумента и установить границу и тип элементов массива.

В начале этой темы отмечалось, что специализация шаблона класса должна проводиться явно. Однако логический вывод аргументов шаблона функции может использоваться для косвенной специализации шаблона класса. Рассмотрим шаблон класса, с помощью которого можно гене-

рировать объект-функцию из указателя функции (см. тему 20 «*Объекты-функции STL*»):

```
template <typename A1, typename A2, typename R>
class PFun2 : public std::binary_function<A1,A2,R> {
   public:
      explicit PFun2( R (*fp)(A1,A2) ) : fp_( fp ) {}
      R operator()( A1 a1, A2 a2 ) const
         { return fp_( a1, a2 );}
   private:
      R (*fp_)(A1,A2);
};
```

(Это упрощенный вариант стандартного шаблона `pointer_to_binary_function` (указатель на бинарную функцию). Он был выбран из-за запутанного синтаксиса его специализации. Хуже не бывает.) Создать экземпляр шаблона напрямую нелегко:

```
bool isGreater( int, int );
std::sort(b, e, PFun2<int,int,bool>(isGreater));  // болезненно
```

В подобных случаях принято предоставлять *вспомогательную функцию* (*helper function*), единственным назначением которой является логический вывод аргументов шаблона с целью автоматической специализации шаблона класса:

```
template <typename R, typename A1, typename A2>
inline PFun2<A1,A2,R> makePFun( R (*pf)(A1,A2) )
   { return PFun2<A1,A2,R>(pf); }
//...
std::sort(b, e, makePFun(isGreater));           // намного лучше...
```

В этой демонстрации силы логического вывода компилятор может вывести типы обоих аргументов, и возвращаемый тип из типа единственного аргумента вспомогательной функции. Эта техника широко применяется в стандартной библиотеке для таких утилит, как `ptr_fun`, `make_pair`, `mem_fun`, `back_inserter` и многих других, вспомогательных функций, упрощающих сложную и чреватую ошибками задачу специализации шаблона класса.

Тема 58 | Перегрузка шаблонов функций

Шаблоны функций могут быть перегружены другими шаблонами функций и нешаблонными функциями. Эта возможность полезна, но ею часто злоупотребляют.

Одним из основных отличий между шаблонами функций и нешаблонными функциями является доступность неявных преобразований фактических параметров. Нешаблонные функции предоставляют широкий диапазон неявных преобразований своих аргументов – от встроенных преобразований (как целочисленное расширение) до пользовательских (неявные одноаргументные конструкторы и операторы преобразования). В случае с шаблонами функций, из-за того что компилятор должен логически выводить аргументы на основании их типов, проводятся только простейшие неявные преобразования, включая квалификацию внешнего уровня (например, `T` в `const T` или `const T` в `T`), ссылку (например, `T` в `T &`) и разложение массива и функции в указатель (например, `T[42]` в `T *`).

Практический эффект этой разницы в том, что шаблоны функций требуют намного более точного соответствия, чем нешаблонные функции. Это может быть хорошо, плохо или просто удивительно. Например, рассмотрим следующее:

```
template <typename T>
void g( T a, T b ) { ... }          // этот g является шаблоном
void g( char a, char b ) { ... }    // этот g - нет
//...
g( 12.3, 45.6 );     // шаблон g
g( 12.3, 45 );       // не шаблон g!
```

Первый вызов с двумя аргументами типа `double` мог бы быть сделан для сопоставления нешаблонной `g` путем неявного преобразования `double`

в char (допускается, но не рекомендуется). Но точное соответствие доступно через создание экземпляра шаблона g с T типа double, поэтому выбирается шаблон. Второй вызов с аргументами типа double и int не будет давать соответствия шаблону g, потому что компилятор не применит предопределенное преобразование из int в double для второго аргумента (или из double в int для первого), чтобы логически вывести для T тип double (или int). Следовательно, вызывается нечлен g с применением так неуместно предопределенных преобразований double и int в char.

Правильный выбор версии функции при наличии множества различных шаблонных и нешаблонных кандидатов – сложный процесс. Многие в других ситуациях надежные компиляторы C++ неправильно выбирают функцию или формируют несоответствующую ошибку. Это значит, что людям, работающим с нашим кодом, будет так же сложно понять, какая версия перегруженного шаблона вызывается. Ради всеобщего блага при использовании перегрузки шаблонов функций необходимо стремиться к максимальной простоте.

«Просто» не значит безыскусно. В теме 57 «*Логический вывод аргументов шаблона*» рассматривалась *вспомогательная* функция, позволяющая обойти трудную и чреватую ошибками специализацию сложного шаблона класса:

```
template <typename A1, typename A2, typename R>
class PFun2 : public std::binary_function<A1,A2,R> {
   // см. реализацию в теме 57 «Логический вывод аргументов шаблона»
...
};
```

Вместо того чтобы заставлять пользователей проводить специализацию этого монстра напрямую, мы предоставили вспомогательную функцию, которая выполнила логический вывод аргументов шаблона и специализацию:

```
template <typename R, typename A1, typename A2>
inline PFun2<A1,A2,R> makePFun( R (*pf)(A1,A2) )
   { return PFun2<A1,A2,R>(pf); }
```

С точки зрения синтаксиса это очень сложный фрагмент кода, но он упрощает работу пользователям. Теперь для функции, объявленной как bool isGreater(int,int), они могут писать не PFun2<int,int,bool>(isGreater), а makePFun(isGreater).

Конечно, нам хотелось бы предоставить механизмы и для унарных функций:

```
template <typename A, typename R>
class PFun1 : public std::unary_function<A,R> {
```

```
    public:
       explicit PFun1( R (*fp)(A) ) : fp_( fp ) {}
       R operator()( A a ) const
          { return fp_( a ); }
    private:
       R (*fp_)(A);
};
```

и для вспомогательной функции:

```
template <typename R, typename A>
inline PFun1<A,R> makePFun( R (*pf)(A) )
    { return PFun1<A,R>(pf);
}
```

Прекрасный пример применения перегрузки шаблонов функций. Он прост в том смысле, что исключает неоднозначность вызываемой версии makePFun (одна для бинарных функций, другая для унарных функций). А одно имя для обеих функций делает эту функцию простой для запоминания и использования.

Тема 59 | Концепция SFINAE

Используя логический вывод аргументов шаблона функции для выбора среди ряда перегруженных шаблонов функций и нешаблонных функций, компилятор может попытаться провести специализацию, которая приведет к ошибке.

```
template <typename T> void f( T );
template <typename T> void f( T * );
//...
f( 1024 ); // создает экземпляр первой f
```

Подстановка ненулевого целого для `T *` во втором шаблоне функции `f` была бы неверной, но попытка подстановки не обусловливает формирования ошибки, если обнаружена правильная подстановка. В данном случае создается экземпляр первого `f`, и ошибки не возникает. Таким образом, получаем концепцию «неудачная подстановка не является ошибкой» (Substitution Failure Is Not An Error – SFINAE), введенную Вандевурдом (Vandevoorde) и Джозаттисом (Josuttis).

SFINAE – важное свойство, потому что без него было бы сложно перегружать шаблоны функций. Без него сочетание логического вывода аргументов и перегрузки давало бы огромное количество недопустимых вариантов использования в наборе перегруженных шаблонов функций. Но SFINAE также ценна как техника метапрограммирования.

Вспомним функцию `IsPtr`, разрабатываемую в теме 52 «*Создание специализации для получения информации о типе*». Чтобы выяснить, является ли неизвестный тип указателем некоторого рода, там применялась частичная специализация. Для достижения того же результата можно воспользоваться SFINAE.

```
typedef char True; // sizeof(True) == 1
```

```
typedef struct { char a[2]; } False;  // sizeof(False) > 1
//...
template <typename T> True isPtr( T * );
False isPtr( ... );

#define is_ptr( e ) (sizeof(isPtr(e))==sizeof(True))
```

Здесь is_ptr позволяет выяснить, является ли выражение указателем, используя логический вывод аргументов шаблона функции и SFINAE. Если выражение e – указатель, компилятор выберет функцию isPtr; в противном случае будет выбрана нешаблонная функция isPtr с пропущенным формальным параметром. SFINAE гарантирует, что попытка сопоставить шаблон isPtr с неуказателем не приведет к ошибке компиляции.

Еще одна особенность этого примера связана с применением оператора sizeof в макросе is_ptr. Обратите внимание, что функции isPtr нигде не определены. Это правильно, потому что они фактически никогда не вызываются. Появление вызова функции в выражении sizeof заставляет компилятор осуществлять логический вывод аргументов и сопоставление функций, но он не вызывает функцию. Оператор sizeof интересует только размер возвращаемого типа функции, которая могла бы быть вызвана. Тогда по размеру возвращаемого типа можно понять, какая функция была выбрана. Если компилятор выбрал шаблон функции, выражение e – указатель.

Нам не пришлось создавать специализации для константных указателей, изменяемых указателей и константных изменяемых указателей, как для функции IsPtr, реализуемой с помощью частичной специализации шаблона класса. При логическом выводе аргументов шаблона функции компилятор проигнорирует cv-квалификаторы «первого уровня» (const и volatile), как и модификаторы ссылок (см. тему 58 «*Перегрузка шаблонов функций*»).

Не надо беспокоиться и о том, что пользовательский тип, имеющий оператор преобразования в тип указателя, будет ошибочно определен как тип указателя. При логическом выводе аргументов шаблона функции компилятор руководствуется очень ограниченным списком преобразований в фактические параметры, и пользовательские преобразования в него не входят.

Обратите внимание на сходство этой техники с применением частичной специализации шаблона с целью выявления информации типа в теме 52 «*Создание специализации для получения информации о типе*». Там «ловушкой» был базовый шаблон, и для выявления необходимых вариантов применялись частичная или полная специализации. Здесь в роли ловушки выступает функция с опущенными формальными параметрами,

а интересующие варианты фиксируются более строгими перегруженными версиями ловушки. Кстати, частичная специализация шаблона класса и перегрузка шаблонов функций очень тесно взаимосвязаны с технической точки зрения. Фактически стандарт определяет алгоритм выбора для одного в терминах другого.

После разбора приведенного выше примера is_ptr практически больше нечего сказать о технической стороне SFINAE. Однако существуют очень неожиданные способы использования этой простой методики для выявления информации о типах и выражениях во время компиляции. Рассмотрим некоторые (не совсем простые) примеры.

Пусть нам требуется определить, является ли неизвестный тип классом:

```
template <typename T>
struct IsClass {
   template <class C> static True isClass( int C::* );
   template <typename C> static False isClass( ... );
   enum { r = sizeof(IsClass<T>::isClass<T>(0))
      == sizeof(True) };
};
```

Точность имеет значение, на этот раз механизм SFINAE инкапсулирован в шаблон класса IsClass и два шаблона функций перегружены как статические члены IsClass. Одна из функций принимает указатель на член класса в качестве формального параметра (см. тему 15 «*Указатели на члены класса – это не указатели*»). Литеральный нуль может быть преобразован в указатель на член класса (даже для шаблона функции). Таким образом, если T – класс, будет выбран первый isClass. Если T не класс, SFINAE проигнорирует ошибочное первое сопоставление и выберет версию isClass с опущенным списком параметров. Как и с is_ptr, проверяя размер возвращаемого типа функции, можно понять, какая функция была выбрана, и, следовательно, определить, является ли T классом.

Второй пример позаимствован у Вандевурда и Джозаттиса. Предположим, необходимо узнать, имеет ли определенный тип класса вложенное имя типа «iterator». (Конечно, это можно применить для любого вложенного имени типа, не только для iterator.)

```
template <class C>
True hasIterator( typename C::iterator const * );
template <typename T>
False hasIterator( ... );
#define has_iterator( C )\
    (sizeof(hasIterator<C>(0))==sizeof(True))
```

Данная функция has_iterator идентична IsClass, но на этот раз осуществляется доступ к вложенному имени неизвестного типа (см. тему 49 «*Устра-*

нение неоднозначности с помощью имени типа»). Если у C есть такой вложенный тип, можно будет преобразовать литеральный нуль в указатель на такой тип. В противном случае будет выбрана ловушка.

И наконец, рассмотрим фокус Андрея Александреску (Andrei Alexandrescu). Можно ли преобразовать T1 в T2, имея два неизвестных типа, T1 и T2? Обратите внимание, что механизм выберет и предопределенное, и пользовательское преобразования:

```
template <typename T1, typename T2>
struct CanConvert {
   static True canConvert( T2 );
   static False canConvert( ... );
   static T1 makeT1();
   enum { r = sizeof(canConvert( makeT1() )) == sizeof(True) };
};
```

Как показала реализация Heap в теме 52 «*Создание специализации для получения информации о типе*», часто гибкость или возможность предоставлять специальные реализации основывается на информации, которая может быть статически выявлена во время компиляции. Используя SFINAE и другие методики метапрограммирования, можно задавать такие вопросы, как: «Является ли этот неизвестный тип указателем на тип класса, имеющий вложенный тип iterator, который может быть преобразован в std::string?».

Тема 60 | Универсальные алгоритмы

Универсальный алгоритм – это шаблон функции, спроектированный таким образом, что может быть легко и эффективно настроен во время компиляции соответственно контексту использования. Рассмотрим шаблон функции, не отвечающий этим стандартам и, следовательно, не являющийся полноценным универсальным алгоритмом:

```
template <typename T>
void slowSort( T a[], int len ) {
    for( int i = 0; i < len; ++i )    // Для каждой пары
        for( int j = i; j < len; ++j )
            if( a[j] < a[i] ) {         // ...если порядок не соблюден...
                T tmp( a[j] );           // ...поменять местами.
                a[j] = a[i];
                a[i] = tmp;
            }
}
```

Этот шаблон может служить для сортировки массива объектов при условии, что для сравнения объектов может быть применен оператор `<` и они могут копироваться. Например, можно сортировать массив объектов `String` из темы 12 «*Присваивание и инициализация – это не одно и то же*»:

```
String names[] = { "my", "dog", "has", "fleece" };
const int namesLen = sizeof(names)/sizeof(names[0]);
slowSort( names, namesLen );       // проведет сортировку...в конце концов!
```

Первое нарекание на `slowSort` (медленная сортировка) – слишком низкая скорость сортировки. Наблюдение верное, но простим ей скорость выполнения $O(n^2)$ и сосредоточимся лучше на универсальности.

Прежде всего можно заметить, что предложенная реализация перестановки в slowSort не идеальна для типа String (и многих других типов, если на то пошло). У класса String есть собственный член swap, одновременно и более быстрый, и более безопасный, чем перестановка, выполняемая путем копирования во временный массив String. Лучшим подходом при реализации будет просто выразить то, что имеется в виду:

```
template <typename T>
void slowSort( T a[], int len ) {
   for( int i = 0; i < len; ++i )    // Для каждой пары
      for( int j = i; j < len; ++j )
         if( a[j] < a[i] )           // ...если порядок не соблюден...
            swap( a[j], a[i] );      // ...поменять местами.
}
```

Здесь по-прежнему не вызывается функция-член swap класса String, но если у автора String «все схвачено», то будет доступна функция-нечлен swap:

```
inline void swap( String &a, String &b )
    { a.swap( b ); }
```

А если в нашем распоряжении нет такого нечлена swap? В этом случае ситуация ничуть не ухудшится, потому что так или иначе в итоге можно вызвать функцию swap стандартной библиотеки, которая делает абсолютно то же самое, что было прописано в исходной версии кода slowSort. Но все равно фактически мы остались в выигрыше, потому что новая реализация slowSort короче, проще и понятнее. И, что еще важнее, если кто-то когда-нибудь реализует эффективную функцию-нечлен swap для String, улучшение будет подхвачено автоматически. С таким обслуживанием кода можно жить.

Теперь рассмотрим сравнение элементов массива с помощью оператора <. Это, наверное, самый распространенный способ сортировки массива (от меньшего к большему). Но может потребоваться сортировать массив в убывающем порядке или как-нибудь иначе. Более того, сортировка может понадобиться массивам объектов, которые или не поддерживают оператор <, или имеют несколько разных кандидатов, подобных оператору «меньше чем». Такой тип уже был представлен в теме 20 «*Объекты-функции STL*»: класс State:

```
class State {
   public:
      //...
      int population() const;
      float aveTempF() const;
      //...
};
```

Подход, рассмотренный в теме 20 «*Объекты-функции STL*», заключался в реализации функций и объектов-функций, которые могли бы использоваться вместо оператора <. Но этот подход эффективен, только если есть универсальный алгоритм принятия такого аргумента:

```
template <typename T, typename Comp>
void slowSort( T a[], int len, Comp less ) {
   for( int i = 0; i < len; ++i )    // Для каждой пары
      for( int j = i; j < len; ++j )
         if( less( a[j], a[i] ) )    // ...если порядок не соблюден...
            swap( a[j], a[i] );      // ...поменять местами.
}
//...
State states[50];
//...
slowSort( states, 50, PopComp() );
```

Если сортировка с помощью функции `slowSort` с оператором `<` так широко применяется, наверное, неплохо было бы перегрузить `slowSort`, чтобы ее можно было вызывать как со специальной операцией сравнения, так и без нее.

Наконец, всегда надо следовать соглашению, и это особенно полезно в случае с универсальными алгоритмами. Можно критиковать `slowSort` и за накладываемое на сортируемый аргумент ограничение – он должен быть массивом, но ведь есть множество других типов контейнеров или структур данных, которые тоже может понадобиться сортировать. Если есть сомнения, посмотрите, как реализована стандартная библиотека:

```
template <typename For, typename Comp>
void slowSort( For b, For e, Comp less ) {
   for( For i( b ); i != e; ++i )       // Для каждой пары
      for( For j( i ); j != e ; ++j )
         if( less( *j, *i  ) )          // ...если порядок не соблюден...
            std::swap( *j, *i  );       // ...поменять местами.
}

template <typename For>
void slowSort( For b, For e ) {
   for( For i( b ); i != e; ++i )       // Для каждой пары
      for( For j( i ); j != e ; ++j )
         if( *j < *i )                  // ...если порядок не соблюден...
            std::swap( *j, *i );        // ...поменять местами.
}
//...
std::list<State> states;
//...
slowSort( states.begin(), states.end(), PopComp() );
```

```
slowSort( names, names+namesLen );
```

Здесь неуклюжий интерфейс массива заменен на более стандартный и гибкий совместимый с STL интерфейс итератора. Теперь можно спокойно вызывать удобный универсальный алгоритм `slowSort`, а не простой шаблон функции.

Один из важных уроков этого примера – сложное ПО практически всегда проектируется группой. По существу, ваш код должен проектироваться с учетом экспертной оценки коллег, но при этом оставаться максимально изолированным от изменений кода, находящегося вне вашего контроля. Улучшенный алгоритм `slowSort` – хороший пример правильного проектирования. Он на максимально возможном абстрактном уровне описывает единственную абсолютно понятную операцию. Точнее, `slowSort` осуществляет алгоритм сортировки, а операции перестановки и сравнения передает другим функциям, которые сделают эту работу лучше. Такой подход позволяет вам, (предполагаемому) эксперту сортировки, расширять свой алгоритм сортировки алгоритмом перестановки для типа элемента, разработанного кем угодно. Вы можете никогда не встречаться, но при правильном подходе к проектированию можете сотрудничать так близко, как если бы работали на одном компьютере. Более того, если в будущем появится улучшенная реализация перестановки, `slowSort` подхватит улучшение автоматически и, вероятно, без вашего ведома. Как никогда раньше, неведение – сила. (Это заставляет нас вспомнить о правильном полиморфном проектировании; см. тему 19 «*Команды и Голливуд*».)

Тема 61 | Мы создаем экземпляр того, что используем

Как в C, так и в C++ объявленную функцию не надо определять, если она не вызывается (или если вы не вызываете ее адрес). Аналогичная ситуация с функциями-членами шаблонов классов: если фактически функция-член шаблона не вызывается, ее экземпляр не создается.

Конечно, это свойство очень полезно для сокращения размера кода. Если шаблон класса определяет много функций-членов, из которых используются только две или три, то неиспользуемые функции не занимают места в программном коде.

Еще более важное следствие из этого правила – можно специализировать шаблоны классов аргументами, которые были бы недопустимыми, если бы создавались экземпляры всех функций-членов. Из этого правила следует, что можно создавать гибкие шаблоны классов, которые могут работать с массой разнообразных аргументов. Даже с теми аргументами, которые могли бы обусловить ошибку при создании экземпляров некоторых функций-членов. Если эти приводящие к ошибке функции не вызываются, их экземпляры не создаются, ошибки не возникает. Это согласуется со многими разделами языка программирования C++, где потенциальные проблемы не являются ошибками до тех пор, пока они не становятся реальными проблемами. В C++ запрещенные помыслы допускаются до тех пор, пока они не начинают реализовываться!

Рассмотрим простой шаблон массива фиксированного размера:

```
template <typename T, int n>
class Array {
   public:
      Array() : a_( new T[n] ) {}
```

```
        ~Array() { delete [] a_; }
        Array( const Array & );
        Array &operator =( const Array & );
        void swap( Array &that ) { std::swap( a_, that.a_ ); }
        T &operator []( int i ) { return a_[i]; }
        const T &operator []( int i ) const { return a_[i]; }
        bool operator ==( const Array &rhs ) const;
        bool operator !=( const Array &rhs ) const
           { return !(*this==rhs); }
    private:
        T *a_;
};
```

Поведение этого контейнера очень похоже на поведение предопределенного массива: те же обычные операции для индексации. Но здесь также предоставляются некоторые высокоуровневые операции, недоступные в предопределенных массивах, такие как перестановка и проверка эквивалентности (опущены операторы отношений, чтобы не занимать много места). Рассмотрим реализацию `operator==`:

```
template <typename T, int n>
bool Array<T,n>::operator ==( const Array &that ) const {
    for( int i = 0; i < n; ++i )
        if( !(a_[i] == that.a_[i]) )
            return false;
    return true;
}
```

Известно, что оба сравниваемых массива имеют одинаковое число элементов, поскольку они одного типа, и размер массива является одним из параметров шаблона. Поэтому требуется только провести попарное сравнение элементов. Если хотя бы в одной паре элементы отличаются, объекты `Array` не равны.

```
Array<int,12> a, b;
//...
if( a == b ) // вызывается a.operator ==(b)
    //...
```

Когда к объектам `Array<int,12>` применяется операция `==`, компилятор создает экземпляр `Array<int,12>::operator==`, компиляция которого проходит без ошибок. Если бы к объектам типа `Array<int,12>` не применялась операция `==` (или `!=`, вызывающая `operator==`), не надо было бы создавать экземпляр этой функции-члена.

Интересная ситуация возникает при создании экземпляра `Array` типа, в котором не определена операция `==`. Например, представим, что тип `Circle` не определяет или не наследует `operator==`:

```
Array<Circle,6> c, d; // нет проблем!
//...
c[3].draw(); // OK
```

Пока все нормально. Операция == ни прямо, ни косвенно не применялась к объекту `Array<Circle,6>`, поэтому экземпляр функции `operator==` не создавался, ошибки нет.

```
if( c == d ) // ошибка!
```

Теперь возникла проблема. Компилятор попытается создать экземпляр `Array<Circle,6>::operator==`, а реализация функции попытается сравнить два объекта `Circle` с помощью несуществующего оператора ==. Ошибка компиляции.

Эта методика обычно используется при проектировании максимально гибких шаблонов классов.

Обратите внимание, что этой идиллии не наблюдается в случае явного создания экземпляра шаблона класса:

```
template Array<Circle,7>; // ошибка!
```

Директива явного создания экземпляра указывает компилятору создать экземпляр `Array` и все его члены с аргументами `Circle` и `7`. В результате при создании экземпляра `Array<Circle,7>:: operator==` возникает ошибка компиляции. Ну, вы сами напросились...

Тема 62 | Стражи включения[1]

В приложениях C++ используется много заголовочных файлов, и часто одни заголовочные файлы включают другие. В таких условиях часто заголовочный файл может быть косвенно включен в компиляцию несколько раз. В больших сложных приложениях нередко один и тот же заголовочный файл встречается сотни раз в одной компиляции. Рассмотрим простой заголовочный файл `hdr2.h`, включающий другой заголовочный файл `hdr1.h`, и заголовочный файл `hdr3.h`, также включающий `hdr1.h`. Если и `hdr2.h`, и `hdr3.h` включены в один и тот же исходный файл, `hdr1.h` будет включен в него дважды. Обычно множественные включения нежелательны и вызывают множество ошибок определения.

По этой причине заголовочные файлы C++ практически везде используют директивы препроцессора. Это предотвращает появление содержимого заголовка в компиляции более одного раза независимо от того, сколько раз фактически включен (`#include`) заголовочный файл. Рассмотрим содержимое заголовочного файла `hdr1.h`:

```
#ifndef HDR1_H
#define HDR1_H
// фактическое содержимое заголовочного файла...
#endif
```

При первом включении (`#include`) заголовочного файла `hdr1.h` в компиляцию символ препроцессора `HDR1_H` не определен, поэтому условная директива препроцессора `#ifndef` («если не определен») делает возможной предварительную обработку директивы `#define` и остального содержимого за-

[1] «Стражи включения» (include guards) – термин, автором которого является Бьярн Страуструп, создатель языка C++.

головочного файла. При втором появлении hdr1.h в той же компиляции символ HDR1_H определен, и #ifndef предотвращает повторное включение содержимого заголовочного файла.

Эта техника будет работать, только если символ, используемый препроцессором для заголовочного файла (в данном случае HDR1_H), ассоциирован только с одним заголовочным файлом (в данном случае hdr1.h). Поэтому важно установить стандарт, простое соглашение о присваивании имен, что обеспечит возможность образовывать имя символа, используемого в страже включения, из имени защищаемого заголовочного файла.

Кроме предотвращения ошибок, стражи включения также помогают повысить скорость компиляции за счет пропуска содержимого заголовочных файлов, которые уже были транслированы. К несчастью, на сам процесс открытия заголовочного файла, проведения оценки #ifndef и поиск завершающего #endif в сложных ситуациях, когда большое количество заголовочных файлов многократно появляется в данной компиляции, может уходить очень много времени. В некоторых случаях избыточные стражи включения могут существенно ускорить дело:

```
#ifndef HDR1_H
#include "hdr1.h"
#endif
```

Вместо того чтобы просто включить (#include) заголовочный файл, мы защищаем включение проверкой того же защитного символа, который задан в заголовочном файле. Такие стражи избыточны, потому что при первом включении заголовочного файла это же условие (в данном случае #ifndef HDR1_H) будет проверено дважды – и перед #include, и в самом заголовочном файле. Однако при последующих включениях избыточный страж предотвратит выполнение директивы #include, предупреждая ненужное открытие и просмотр заголовочного файла. Избыточные стражи включения не так распространены, как одиночные, но в некоторых случаях использование первых может существенно улучшить время компиляции больших приложений.

Тема 63 | Необязательные ключевые слова

Некоторые ключевые слова с точки зрения языка программирования C++ абсолютно необязательны, хотя по другим соображениям их наличие или отсутствие может быть спорным.

Чаще всего источником путаницы становится необязательное использование virtual в функции-члене производного класса, переопределяющей виртуальную функцию-член базового класса.

```
class Shape {
   public:
      virtual void draw() const = 0;
      virtual void rotate( double degrees ) = 0;
      virtual void invert() = 0;
      //...
};
class Blob : public Shape {
   public:
      virtual void draw() const;
      void rotate( double );
      void invert() = 0;
      //...
};
```

Функция-член Blob::draw переопределяет функцию draw (рисовать) базового класса и, таким образом, является виртуальной. Ключевое слово не оказывает никакого влияния на смысл программы, и его совершенно спокойно можно опустить. Всеобщее заблуждение в том, что пропуск ключевого слова virtual сделает невозможным дальнейшее переопределение в последующих производных классах. Это не так.

```
class SharpBlob : public Blob {
```

```
    public:
     void rotate( double ); // переопределяет Blob::rotate
     //...
};
```

Обратите внимание, что ключевое слово virtual также можно опустить и при переопределении чисто виртуальной функции, как в Blob::invert. Наличие или отсутствие virtual в переопределяющей функции производного класса полностью произвольно и не влияет на смысл программы. Никоим образом.

Мнения по поводу того, хорошо ли опускать ключевое слово virtual в переопределяющей функции производного класса, разделились. Некоторые специалисты утверждают, что незначащее virtual помогает задокументировать природу функции производного класса для человека – читателя программы. Другие называют это лишней тратой усилий и считают, что это может привести к «случайному» превращению непереопределяющей функции производного класса в виртуальную. Неважно, какого мнения придерживаться, главное быть последовательным – или использовать ключевое слово virtual во всех переопределяющих функциях производного класса, или везде опускать его.

Ключевое слово static можно опускать при объявлении членов operator new, operator delete, и их версий для массивов (см. тему 36 «*Индивидуальное управление памятью*»), потому что эти функции неявно статические.

```
class Handle {
    public:
       void *operator new( size_t );          // неявно статическая
       static void operator delete( void * );
       static void *operator new[]( size_t );
       void operator delete[]( void * );      // неявно статическая
};
```

Некоторые специалисты утверждают, что лучше всего быть точным и всегда явно объявлять эти функции статическими. Другие думают, что если пользователь программы на C++ не знает о неявной статичности этих функций, он не должен использовать или обслуживать код. Указывать здесь ключевое слово static – пустая трата сил. Программа не место для размещения шпаргалок по семантике языка. Как и с virtual, какую бы позицию вы ни занимали относительно static, важно быть последовательным. Или все эти четыре функции должны явно объявляться статическими, или ни одна из них.

В заголовке шаблона ключевые слова typename и class являются взаимозаменяемыми. Они обозначают, что параметр шаблона является именем типа, и ничем не отличаются. Однако в коде опытных программистов на

C++ typename сообщает читателю-человеку, что аргумент шаблона может быть любого типа, а class — что аргумент типа должен быть классом.

```
template <typename In, typename Out>
Out copy( In begin, In end, Out result );

template <class Container>
void resize( Container &container, int newSize );
```

В прежние времена при помощи ключевого слова register программист мог «намекнуть» компилятору, что переменная (по мнению программиста) будет активно использоваться и поэтому должна быть помещена в регистр. Также не допускалось получение адреса переменной, объявленной с ключевым словом register. Однако вскоре создатели компиляторов поняли, что их коллеги, занимающиеся разработкой программ, не имели никакого понятия о том, какие переменные должны храниться в регистре. Поэтому теперь компиляторы полностью игнорируют такие предложения программистов. В C++ ключевое слово register вообще никак не влияет на смысл программы и обычно не меняет ее эффективности.

Ключевое слово auto может служить для обозначения того, что автоматическая переменная (аргумент функции или локальная переменная) является автоматической. Не стоит с ним возиться.

Чтобы быть до конца честным, отмечу, что и register, и auto могут устранять неоднозначность синтаксиса, когда код написан особенно скудно. Правильным подходом в таких случаях будет написать лучший код, позволяющий обойтись без этих ключевых слов.

Библиография

1. Alexandrescu, Andrei. *Modern C++ Design*. Addison-Wesley, 2001.[1]
2. Dewhurst, Stephen C. *C++ Gotchas*. Addison-Wesley, 2003.
3. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.[2]
4. Josuttis, Nicolai M. *The C++ Standard Library*. Addison-Wesley, 1999.
5. Meyers, Scott. *Effective C++*, Third Edition. Addison-Wesley, 2005.
6. Meyers, Scott. *Effective STL*. Addison-Wesley, 2001.
7. Meyers, Scott. *More Effective C++*. Addison-Wesley, 1996.
8. Sutter, Herb. *Exceptional C++*. Addison-Wesley, 2000.[3]
9. Sutter, Herb. *More Exceptional C++*. Addison-Wesley, 2002.[4]
10. Sutter, Herb. *Exceptional C++Style*. Addison-Wesley, 2005.
11. Sutter,Herb, and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, 2005.[5]
12. Vandevoorde, David, and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley, 2003.
13. Wilson, Matthew. *Imperfect C++*. Addison-Wesley, 2005.

---

[1] А. Александреску «Современное проектирование на C++», Вильямс, 2002.

[2] Э. Гамма, Р. Хельм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2001.

[3] Г. Саттер «Решение сложных задач на C++», Вильямс, 2002.

[4] Г. Саттер «Новые сложные задачи на C++», Вильямс, 2005.

[5] Г. Саттер, А. Александреску «Стандарты программирования на C++», Вильямс, 2005.

Алфавитный указатель

Символы

A

C

E,

G

H

J

O

P

Q

R

S

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ч

Ш

Э

Я